Titre: Title:	Conception et validation d'algorithmes parallèles pour l'adaptation de maillage
Auteur: Author:	Sébastien Laflamme
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Laflamme, S. (2004). Conception et validation d'algorithmes parallèles pour l'adaptation de maillage [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7216/

UNIVERSITÉ DE MONTRÉAL

CONCEPTION ET VALIDATION D'ALGORITHMES PARALLÈLES POUR L'ADAPTATION DE MAILLAGE

SÉBASTIEN LAFLAMME
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JANVIER 2004

Your file Votre référence
ISBN: 0-612-89214-X
Our file Notre référence
ISBN: 0-612-89214-X

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

CONCEPTION ET VALIDATION D'ALGORITHMES PARALLÈLES POUR L'ADAPTATION DE MAILLAGE

présenté par: LAFLAMME Sébastien

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

M. TRÉPANIER Jean-Yves, Ph.D., président

M. GUIBAULT François, Ph.D., membre et directeur de recherche

M. ROY Robert, Ph.D., membre et codirecteur de recherche

M. DAGENAIS Michel, Ph.D., membre

REMERCIEMENTS

Je remercie chaleureusement mon directeur de recherche, François Guibault et mon codirecteur de recherche, Robert Roy, pour m'avoir épaulé tout au long de la mise en oeuvre de ce projet ainsi que pour le support financier octroyé durant les deux années qui se sont écoulées lors de sa réalisation.

Je tiens également à remercier tout particulièrement Julien Dompierre, assistant de recherche à l'école Polytechnique de Montréal et principal développeur de OORT, la librairie séquentielle sur laquelle IP-OORT repose. Julien m'a aidé à maintes reprises au niveau du développement de IP-OORT et OORT. Son aide fut également très précieuse lors de la rédaction du présent document (et de plusieurs autres) à l'aide de LaTeX.

Je tiens également à remercier Paul Labbé, ex-assistant de recherche au CERCA (Centre de REcherche en Calcul Appliqué), qui m'a fourni une aide importante en quelques occasions principalement pour retracer des « bugs » informatiques complexes. Un remerciement bien spécial à Monaco, le chien de Paul, qui m'a souvent désennuyé lors de certains après-midi pénibles !

Je remercie également Éric Malouin qui a développé le premier prototype de IP-OORT.

Je remercie également tous les membres de ma famille ainsi que mes amis pour m'avoir enduré (!) durant ces deux années.

Je tiens finalement à remercier tous les chercheurs, professeurs, étudiants et employés que j'ai côtoyés durant ces deux années au CERCA et à l'école Polytechnique qui ont fait en sorte que l'on puisse travailler dans une ambiance intéressante.

RÉSUMÉ

Les processus de simulation numérique requièrent un bon contrôle sur l'erreur. Les méthodes d'adaptation de maillage permettent justement d'avoir ce contrôle en fournissant des mécanismes permettant d'adapter un maillage à une solution dans le but d'en améliorer la précision. Le processus de simulation numérique est donc itératif et consiste en une répétition des étapes de calculs de solution et d'adaptation de maillage en fonction de la solution de façon à converger vers un maillage et une solution optimaux. Dans ce schéma, l'adaptation de maillage est une étape complètement indépendante des autres avec ses propres contraintes et particularités. Or, bien que plusieurs travaux aient été faits pour paralléliser les phases de génération de maillage et de résolution (résoluteurs par éléments finis, volumes finis ou différences finies), bien peu l'ont été au niveau de l'adaptation de maillage qui risque alors d'être un goulot d'étranglement dans le schéma global de simulation.

Le présent projet consiste donc à travailler sur un logiciel d'adaptation de maillage parallèle qui puisse s'intégrer dans le schéma global de simulation numérique. Le logiciel en question est basé sur une version séquentielle déjà existante et fonctionnelle et en réutilise les algorithmes par un mécanisme d'héritage de classes. Dans une implantation préliminaire, l'application parallèle ne supporte que le lissage géométrique (déplacement des sommets) sur les maillages structurés. Les algorithmes de déplacement de sommets en séquentiel sont directement utilisés dans la version parallèle. Cette dernière se contente d'ajouter les fonctionnalités requises au parallélisme tel le partitionnement de domaine, la cohérence des frontières et toutes les communications entre les différents processeurs.

Le projet se divise en deux phases. La première phase consiste à évaluer un premier prototype de logiciel d'adaptation de maillage en parallèle pour en estimer les performances et les principaux problèmes. La deuxième phase décrit les changements apportés à l'ar-

chitecture et aux algorithmes afin de résoudre les problèmes identifiés.

Les résultats obtenus avec la nouvelle version sont très encourageants. La qualité des maillages s'est améliorée et il n'y a pas de différences majeures de qualité entre la version parallèle et la version séquentielle. En outre, les performances de l'application ont été grandement améliorées et l'application parallèle donne maintenant des accélérations beaucoup plus raisonnables.

Malheureusement, certains problèmes subsistent. Par exemple, un problème au niveau de l'équilibre de la charge des processeurs limite les gains de performance en parallèle. En outre, il a été établi que la difficulté à définir des fonctions de poids représentant fidèlement la charge de travail associée à chaque sommet et la faible granularité du partitionnement constitue un obstacle à la possibilité de repartitionner dynamiquement le domaine de façon efficace. Finalement, une analyse de la possibilité d'intégrer les maillages non structurés est faite et celle-ci démontre que les algorithmes parallèles, initialement conçus pour les maillages structurés, se prêtent mal à une intégration des maillages non structurés. De nouveaux algorithmes devraient donc être développés pour le support de ce type de maillage.

En résumé, le projet a permis d'améliorer considérablement le prototype d'adaptation de maillage en parallèle, mais de nombreux problèmes subsistent. Plusieurs nouvelles pistes de recherche sont ainsi définies.

ABSTRACT

A numerical simulation process needs to provide efficient error control. Mesh adaptation is a good way to provide this control as it provides mechanisms to adapt the mesh to a given solution in order to get a better solution. The numerical simulation process is iterative. A resolution step is usually followed by an adaptation step which will allow a new resolution step which, hopefully, will provide a better solution. However, mesh adaptation is a fully independent procedure which has its own set of constraints and while significant efforts have been made on to accelerate the speed of solvers using parallelization techniques, very few have been made on mesh adaptation which could then be the bottle neck of the simulation scheme.

The project described in this document aims to develop a parallel mesh adapter based on an existing sequential mesh adaptation software. At first, only structured meshes are considered in the parallel version, so only the vertices relocation algorithm is considered. The sequential version algorithms are directly used in the parallel version which simply define partitions on which each processor applies them. The parallel version provides algorithms to partition the mesh, to communicate information between sub-domain boundaries and to synchronize processes.

This project is divided in two phase. First, an evaluation of the parallel mesh adapter prototype is done and the main problems are identified. Then, the changes done to the architecture and to the algorithms of the parallel mesh adapter are presented.

Results obtained with the new version are encouraging. Mesh quality is improved and the parallel version produce the same quality of meshes than the sequential version. Also, the execution times are much better.

However, some problems remain. For example, a load balancing problem is still present, which limits the performance obtained with the parallel version. Also, the difficulty to develop a weight function associated with every vertex that accurately represents

the amount of work to do with this vertex and a too big mesh granularity limit the possibility of implementing an efficient dynamic repartitioning algorithm which could solve the load balancing problem. Finally, the possibility of extending the functionalities of the parallel version in order to work on unstructured meshes is evaluated. Since the algorithms were specifically developed for structured meshes, it seems that the support of unstructured meshes would require the development of completely new algorithms.

In short, this project considerably improved the parallel mesh adapter prototype. However, numbers on problems remain and new track of research are defined.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

INTRODUCTION

Contexte du projet

Le projet de recherche présenté dans ce mémoire s'inscrit dans le cadre d'un vaste projet de développement d'outils automatiques de simulation numérique dans le domaine de la mécanique des fluides. Plus particulièrement, le projet vise à automatiser et à optimiser les processus de génération et d'adaptation de maillage afin d'améliorer la qualité des simulations numériques. Ce projet est mené conjointement par le GRMIAO (Groupe de Recherche en Mathématiques de l'Ingénierie Assistée par Ordinateur de l'École Polytechnique de Montréal)[1]), le CERCA (CEntre de Recherche en Calcul Appliqué) et GE Hydro. GE Hydro est une division de General Electric qui se spécialise dans la conception, la fabrication et la vente de turbines hydrauliques destinées à des projets hydroélectriques à travers le monde. Chacun de ces projets est unique en soit et nécessite la fabrication de turbines qui sont conçues et optimisées de manière spécifique au site en tenant compte des paramètres particuliers de celui-ci (hauteur de chute, débit d'eau, ...).

La technique employée pour concevoir une turbine hydraulique consiste à concevoir un premier design, à valider ce design et à y apporter les correctifs nécessaires. C'est un processus itératif, les étapes de conception et de validation se succédant jusqu'à ce que les résultats soient satisfaisants. Traditionnellement, pour valider le design, la principale technique employée consiste à construire des prototypes en modèles réduits et de faire des tests en laboratoire pour évaluer les performances de la turbine. Il s'agit là d'un procédé très coûteux en temps et nécessitant des ressources considérables. Les simulations numériques représentent une alternative avantageuse à ce procédé puisqu'elles

[1] http ://www.polymtl.ca/grmiao/grmiao/

sont beaucoup moins coûteuses et plus rapides. Malheureusement, ces simulations restent pour le moment moins précises que les tests en laboratoire. Dans le cas de GE Hydro, les simulations numériques sont principalement utilisées lors de la phase de design et les résultats sont ensuite validés par des tests en laboratoire sur des prototypes avant de passer à la phase de fabrication. Le projet mené en collaboration avec l'École Polytechnique de Montréal vise à améliorer la performance des simulations numériques et à réduire la nécessité du recours aux tests en laboratoire.

La figure 1.1 présente le schéma global de simulation numérique. On y voit les différents modules utilisés au cours du processus ainsi que les entrées et les sorties de ces modules. Le point de départ de toute simulation numérique est naturellement la conception du modèle géométrique. À l'aide d'un modeleur géométrique comme I-DEAS[2] distribué par la compagnie EDS[3] par exemple, le concepteur modélise la géométrie du domaine sur lequel la simulation numérique sera faite. La géométrie sera ensuite maillée à l'aide d'un générateur de maillage, aussi appelé un mailleur. Le maillage obtenu permettra ensuite de résoudre le problème par la méthode des éléments finis, des volumes finis ou des différences finies, selon le type de résoluteur utilisé. Ces trois différentes méthodes ont toutes un point en commun. Elles utilisent un maillage, donc une discrétisation de l'espace, afin de résoudre numériquement le problème. On obtient donc une solution numérique dont la précision dépendra du maillage utilisé. Or, comment peut-on prévoir à l'avance la solution de façon à générer un maillage adéquat pour cette solution ? L'approche préconisée pour résoudre ce problème est une approche itérative qui consiste à adapter le maillage en fonction de la solution obtenue. Ceci permet ensuite de calculer une nouvelle solution plus précise à l'aide d'un maillage mieux adapté au problème. Ce processus peut se répéter jusqu'à ce qu'on atteigne un certain degré de convergence.

[2]www.eds.com/products/plm/ideas/
[3]www.eds.com

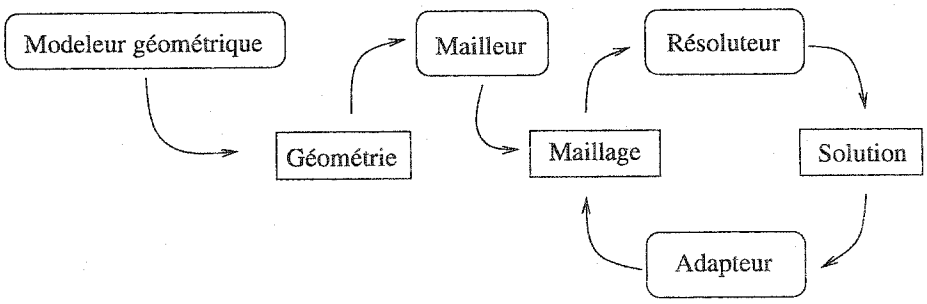


FIGURE 1.1: Schéma global du processus de résolution numérique

Problématique à résoudre

Ce projet a déjà donné des résultats intéressants en mode séquentiel tel que présentés dans les travaux de Vu *et al.* (2000) et de Guibault *et al.* (1999). Toutefois, le temps d'exécution du processus d'adaptation de maillage est devenu un facteur important non seulement dans le cadre de ce projet, mais également dans plusieurs domaines liés à la mécanique des fluides. En effet, l'adaptation de maillage est une étape parmi plusieurs lors du processus complet de simulation numérique mais, bien que des efforts considérables aient été investis dans la parallélisation des résoluteurs (on trouve plusieurs résoluteurs industriels fonctionnant en mode parallèle comme par exemple CFX-5[4] et CFX-TASCFLOW[5]), relativement peu l'ont été dans la parallélisation du processus d'adaptation. De ce fait, l'adaptation de maillage peut devenir un goulot d'étranglement à l'intérieur du schéma global de simulations numériques.

Le sujet du présent mémoire vise donc la conception d'algorithmes parallèles pour l'adaptation de maillage. Il y sera question des efforts faits dans le but de paralléliser une application d'adaptation de maillage existante afin d'en accélérer l'exécution tout en préservant la qualité des maillages obtenus en sortie.

[4]http ://www-waterloo.ansys.com/cfx/products/
[5]http ://www.aeat.com/cfx/WTASCflow.html

Les principaux problèmes à résoudre pour la réalisation concerne le partitionnement du domaine et le maintien de la cohérence de l'information. En effet, pour paralléliser l'application, il faudra d'abord diviser le domaine en sous-domaines. Chaque processeur pourra alors travailler sur son propre sous-domaine. Toutefois, comment s'assurer que les partitions sont optimales et que chaque processeur ait une quantité de travail équivalente à effectuer ? Comment maintenir la cohérence du maillage global lorsque le domaine est ainsi partitionné ? Il s'agit de bien définir les frontières des sous-domaines ainsi que les différentes étapes de synchronisation et de communication entre les processus qui permettront à l'ensemble des données de demeurer cohérent tout en maximisant la performance obtenue. Ces problèmes ne sont pas triviaux. Par exemple, il faut souvent faire des compromis entre la qualité d'un partitionnement en terme de répartition de la tâche et la quantité de communication requise liée à la gestion des frontières.

Les principaux requis du système à développer sont les suivants :

1. *Obtenir des maillages valides.* Il est primordial que la version parallèle du système produise des maillages de qualité équivalente à la version séquentielle, car il ne sert à rien d'avoir une application plus performante si celle-ci produit des résultats erronés.
2. *Obtenir un gain de performance appréciable.* Le gain de performance obtenu en parallèle doit être le meilleur possible.
3. *Fonctionnement sur les maillages structurés.* Dans un premier temps, seuls les maillages structurés seront supportés par l'application parallèle. Ce choix est expliqué dans le chapitre 2.
4. *Réutilisation des algorithmes séquentiels.* L'application séquentielle étant toujours en développement, il est essentiel d'en utiliser directement les algorithmes dans la version parallèle afin de bénéficier des améliorations qui pourraient leur être apportées.

Hypothèses de départ

Les principaux requis ayant été identifiés, il convient maintenant d'établir les hypothèses de départ du projet :

1. *Existence d'un résoluteur parallèle.* Il a déjà été établi que l'adaptation de maillage s'inscrit dans un processus complet de simulation numérique. En outre, l'adaptation de maillage a été présentée comme un goulot d'étranglement étant donnée l'existence de résoluteurs parallèles et le manque de remailleurs parallèles. Ainsi, une des hypothèses en vue du projet concerne l'existence de résoluteurs parallèles. On en a d'ailleurs mentionnés quelques uns précédemment.
2. *Stabilité des algorithmes séquentiels.* Les algorithmes séquentiels qui seront utilisés sont fonctionnels et présentent certaines caractéristiques importantes qui sont présentées ici. D'abord, il est à noter que puisque la parallélisation se fait pour les maillages structurés, c'est l'algorithme de déplacement de sommets qui est utilisé (des explications sur ce choix sont présentées ultérieurement). Voici quelques hypothèses importantes au sujet de cet algorithme :
   (a) *Déplacement valide.* Lorsqu'un sommet est déplacé, le maillage reste valide, car l'algorithme le vérifie. Dans le cas où le déplacement produirait un maillage non valide, l'algorithme laisse le sommet à sa position originale.
   (b) *Connaissance des voisins.* Pour le bon fonctionnement de l'algorithme, chaque sommet doit connaître la position de tous ses voisins, c'est-à-dire tous les sommets partageant un même élément.
   (c) *Statistiques sur le déplacement et convergence.* Lors du processus, des statistiques sur le déplacement sont maintenues afin d'afficher la progression du travail fait à l'usager et de vérifier la convergence.
3. *Maillage d'entrée valide.* Le maillage en entrée est valide et repose sur une géométrie et une topologie correctement définies.

Objectifs du projet

Les objectifs du projet sont résumés ici. Dans un premier temps, il convient de préciser qu'un premier prototype de remailleur parallèle a été conçu au CERCA. Toutefois, ce prototype comporte plusieurs problèmes tant du côté de la performance que du côté de la qualité des maillages obtenus en sortie. L'objectif général du projet est donc d'améliorer ce premier prototype pour en faire une application robuste et performante. De façon plus précise, les objectifs sont :

1. Analyser le premier prototype de remailleur parallèle pour en faire ressortir les principales caractéristiques de performance, de fiabilité et de robustesse tout en identifiant ses principales lacunes. C'est la première étape avant de développer de nouveaux algorithmes.
2. Concevoir de nouveaux algorithmes afin de régler les problèmes identifiés lors de l'analyse du prototype.
3. Examiner la possibilité d'ajouter le support des maillages non structurés. Pour l'instant, seuls les maillages structurés sont supportés. Il sera donc intéressant et pertinent d'analyser la possibilité d'y ajouter les maillages non structurés. Les principaux problèmes et obstacles à surmonter dans ce cas seront analysés.

Cadre de travail

Cette section présente le cadre général dans lequel le sujet du présent mémoire s'inscrit ainsi que les différentes contraintes à respecter.

Le logiciel d'adaptation de maillage en parallèle à développer devra s'intégrer dans un cadre de travail bien particulier. Au GRMIAO (et auparavant au CERCA), plusieurs outils de simulations numériques s'inscrivant dans le cadre du schéma de résolution de la

figure 1.1 ont été développés. À la base de ces différents outils, on retrouve la librairie de classes C++ **PIRATE** qui fournit un ensemble de classes d'objets pour traiter et manipuler une foule de données pour les différentes phases de la simulation numérique, soit la phase de résolution à l'intérieur de résoluteurs (variables, conditions frontières, opérateurs différentiels, ...), les phases de préparation des données (géométries, maillages, fonctions de concentration) et la phase d'analyse (solution, estimation d'erreur, ...). **PIRATE** contient notamment des structures de données et des méthodes de traitement pour gérer des géométries et des topologies, des maillages et des solutions définies sur ces maillages avec différentes méthodes d'interpolation. **PIRATE** définit également un format d'échange de données et fournit les mécanismes requis pour la sauvegarde et la lecture de fichiers. Une description plus détaillée de la librairie **PIRATE** peut être trouvée dans les travaux de Labbé *et al.* (2000), de Ozell *et al.* (1995) et de Guibault *et al.* (1995). Le manuel de l'usager[6] et la description du format de fichier[7] utilisé par **PIRATE** sont disponibles en ligne.

Plusieurs outils de simulation numérique ont été conçus en étant basés sur cette librairie. On retrouve notamment un générateur de maillage, maille (Labbé *et al.* (2001)), et un logiciel d'adaptation de maillage, **OORT** (Dompierre et Labbé (1999)). Le logiciel d'adaptation de maillage en parallèle dont traite le présent mémoire sera directement basé sur **OORT**. Il travaillera donc en étroite collaboration avec **OORT** pour les algorithmes d'adaptation et **PIRATE** pour la gestion des données.

Au chapitre de l'adaptation, le module d'adaptation de maillage, **OORT**, a déjà donné plusieurs bons résultats. Par exemple, les travaux de Vu *et al.* (2000) montre les résultats obtenus lors du couplage de **OORT** avec un résoluteur commercial, CFX-TASCFLOW. Des travaux ont également été fait avec CFX-5. La version parallèle devrait donc nous

[6] www.polymtl.ca/grmiao/grmiao/Pir/doc/manuel/usr/usr/
[7] www.polymtl.ca/grmiao/grmiao/Pir/doc/manuel/fic/fic/

permettre d'obtenir les mêmes résultats ou, à tout le moins, des résultats comparables. Normalement, la version parallèle devrait pouvoir fonctionner partout où la version séquentielle fonctionne et être couplée à des résoluteurs tels CFX-TASCFLOW et CFX-5.

Présentation du mémoire

Le présent mémoire est divisé en quatre chapitres. Dans le premier chapitre, une revue des concepts de base est faite. Les questions de génération de maillage, d'adaptation de maillage avec les différentes méthodes d'adaptation possibles, de partitionnement de domaine et de parallélisme sont abordées afin de faire une revue de ce qui s'est fait dans ces différents domaines. Le second chapitre présente une analyse complète du prototype initial de remailleur parallèle. Une attention particulière est apportée à la performance et à la qualité des maillages obtenus en sortie. Le troisième chapitre présente les nouveaux algorithmes développés afin de résoudre les problèmes identifiés au deuxième chapitre. Les changements apportés, tant au code séquentiel qu'au code parallèle y sont également présentés. Le dernier chapitre présente les résultats obtenus avec une analyse et une discussion sur ces résultats. Est-ce que les nouveaux algorithmes permettent réellement de résoudre les problèmes ? Finalement, en conclusion, une revue du travail fait et du travail à faire dans le futur est présentée.

CHAPITRE 1

REVUE DES CONCEPTS

1.1 Introduction

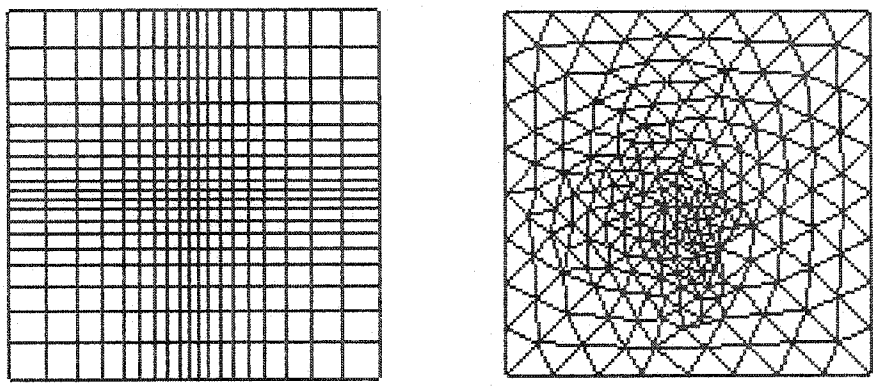
Ce chapitre vise à présenter une définition claire des concepts mis en jeu dans le domaine des maillages et du parallélisme. Cela servira de base au reste du travail. On retrouve d'abord les concepts liés directement au domaine des maillages comme la génération de maillage et l'adaptation de maillage. Ensuite, on retrouve les concepts liés au traitement parallèle. Différents algorithmes de partitionnement de domaine sont présentés ainsi que certains algorithmes d'adaptation de maillage en parallèle.

Toutefois, avant de passer à la définition de ces concepts, il est nécessaire d'établir clairement ce qu'est un maillage. À sa plus simple expression, un maillage n'est qu'une partition de l'espace. Mathématiquement, on peut représenter le maillage $\mathcal{M}$ comme étant l'ensemble suivant :

$$\mathcal{M} = \left\{ N, \{X_i\}_{i=1}^{N}, M, \{\nu_j\}_{j=1}^{M} \right\}$$

où N est le nombre de sommets du maillage, $\{X_i\}_{i=1}^{N}$ sont les N coordonnées du maillage, M est le nombre d'éléments du maillage et $\{\nu_j\}_{j=1}^{M}$ sont les connectivités des éléments du maillage. Essentiellement, un maillage est donc un ensemble de sommets définis par leurs coordonnées et un ensemble d'éléments reliant ces sommets.

La figure 1.2 montre un exemple de géométrie 2D discrétisée à l'aide d'un maillage structuré et d'un maillage non structuré. La principale différence entre les deux types de



(a) Maillage structuré (b) Maillage non structuré

FIGURE 1.2: Exemples de maillages 2D

maillage est que, dans le cas des maillages structurés, la connectivité est implicite. En effet, étant donné un sommet (i, j) (ou (i, j, k) en 3D) on peut retrouver automatiquement ses voisins par des formules simples. Il suffit de prendre les index voisins ($i \pm 1$, $j \pm 1$ ou $k \pm 1$). De plus, il est important de noter que dans un maillage structuré, chaque sommet, à l'exception des sommets frontières, a exactement le même nombre de voisins alors que dans un maillage non structuré, ce nombre peut varier. Il existe également un troisième type de maillage. Ce sont les maillages hybrides. Ce type de maillage est une combinaison des deux autres. Ainsi, certaines régions sont maillées de façon structurée et d'autres de façon non structurée. Cette technique est parfois utilisée de façon à conserver toute la flexibilité des maillages non structurés à l'intérieur du domaine de calcul tout en conservant les caractéristiques de régularité importantes des maillages structurés près des frontières du domaine.

1.2 Génération de maillage

Dans le cadre d'une simulation numérique, la première étape, comme on peut le voir à la figure 1.1, est la génération du modèle géométrique représentant le domaine sur

lequel la solution numérique devra être calculée. Une fois ce travail fait, l'étape suivante est la génération du maillage. En effet, toutes les techniques de résolution numérique (éléments finis, volumes finis, ...) reposent sur une discrétisation du domaine, c'est-à-dire un maillage. Ainsi, avant de passer dans un résoluteur, il faudra d'abord générer un maillage à partir de la géométrie à analyser.

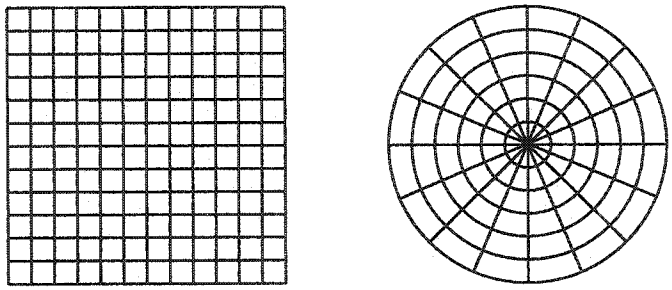
Il existe plusieurs techniques pour générer des maillages. Certaines techniques sont destinées à la création de maillages structurés, alors que d'autres permettent de créer des maillages non structurés. Le livre de Frey et George (1999) constitue une excellente référence en la matière. On y présente plusieurs concepts liés aux maillages et on y fait une revue des techniques de génération de maillage. Les sous-sections suivantes présentent un résumé des principales techniques pour générer des maillages structurés et non structurés.

1.2.1 Génération de maillage structuré

Le maillage structuré est construit de telle sorte que la connectivité est implicite, ce qui signifie que chaque sommet possède le même nombre de voisins et que ceux-ci peuvent être facilement retrouvés grâce aux index. Généralement, on peut classer les maillages structurés en deux types.

Premièrement, les maillages structurés réguliers sont des maillages qui sont généralement basés directement sur des systèmes de coordonnées classiques tels que les systèmes cartésien et polaire par exemple. La figure 1.3 en montre des exemples. Le principal problème de ces maillages est qu'ils ne peuvent pas épouser pas les frontières particulières d'un domaine.

On retrouve également des maillages curvilignes. Ces maillages sont construits par le



(a) Système cartésien (b) Système polaire

FIGURE 1.3: Exemples de maillages réguliers structurés selon un système de coordonnées.

biais d'un transformation entre un espace paramétrique et un espace physique de la forme :

$$(x, y) = (f(u, v), g(u, v))$$

Ces types de maillages permettent d'intégrer des domaines ayant une frontière quasi quelconque. Toutefois, on doit être en mesure de diviser la frontière en deux paires de courbes correspondant aux valeurs limites des paramètres u et v, ce qui, dans bien des cas, peut s'avérer difficile. Une transformation algébrique souvent utilisée pour générer le maillage est l'interpolation transfinie bivariée. Cette transformation consiste à faire une interpolation bivariée de la surface à partir de ses quatre frontières et les lignes de maillage correspondent à des isovaleurs des paramètres u et v. Mathématiquement,

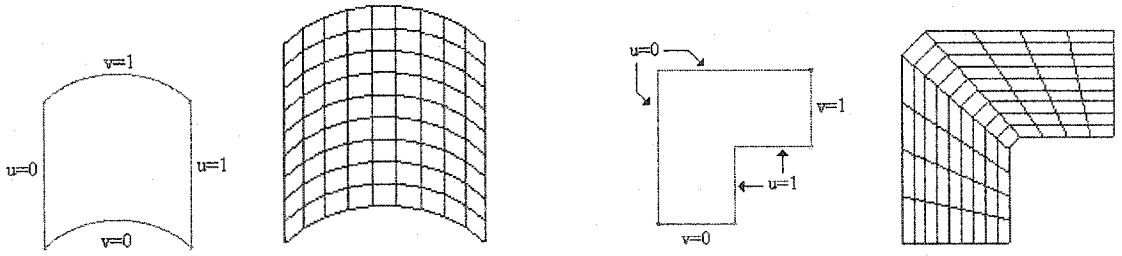
l'interpolation transfinie bivariée s'exprime de la façon suivante :

$$\begin{aligned}\overrightarrow{X} = & \begin{bmatrix} 1-u & u \end{bmatrix} \begin{bmatrix} \overrightarrow{P_{0,v}} \\ \overrightarrow{P_{1,v}} \end{bmatrix} + \begin{bmatrix} \overrightarrow{P_{u,0}} & \overrightarrow{P_{u,1}} \end{bmatrix} \begin{bmatrix} 1-v \\ v \end{bmatrix} \\ & - \begin{bmatrix} 1-u & u \end{bmatrix} \begin{bmatrix} \overrightarrow{P_{0,0}} & \overrightarrow{P_{0,1}} \\ \overrightarrow{P_{1,0}} & \overrightarrow{P_{1,1}} \end{bmatrix} \begin{bmatrix} 1-v & v \end{bmatrix}\end{aligned}$$

où

$$\begin{cases} \overrightarrow{P_{0,v}} \text{ correspond à la frontière } u = 0, \text{ évaluée en v} \\ \overrightarrow{P_{1,v}} \text{ correspond à la frontière } u = 1, \text{ évaluée en v} \\ \overrightarrow{P_{u,0}} \text{ correspond à la frontière } v = 0, \text{ évaluée en u} \\ \overrightarrow{P_{u,1}} \text{ correspond à la frontière } v = 1, \text{ évaluée en u} \end{cases}$$

La figure 1.4 montre deux géométries avec leurs paires de frontières ainsi que le résultat obtenu par cette technique.



(a) Géométrie à mailler (b) Maillage obtenu (c) Géométrie à mailler (d) Maillage obtenu

FIGURE 1.4: Exemples d'interpolations transfinies bivariées.

Il existe plusieurs autres méthodes pour générer des maillages structurés. La méthode elliptique, par exemple, est basée sur la résolution d'un système d'équations aux dérivées partielles. L'équation de Laplace est fréquemment utilisée dans ce cas, ce qui donne le système d'équations suivant :

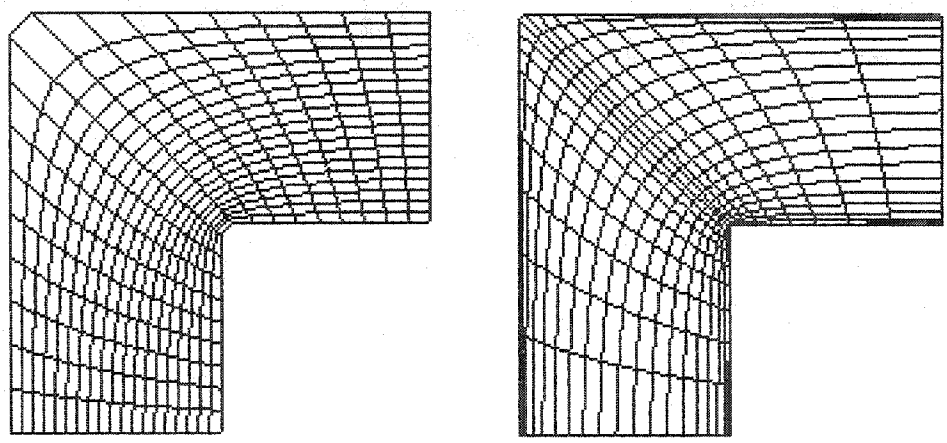
$$\nabla^2 \epsilon_i = 0$$

où i prend les valeurs de 1 à n, avec n étant la dimension du problème. Les maillages obtenus par cette technique possèdent certaines propriétés très intéressantes qui sont

héritées de l'équation de Laplace comme les propriétés de lissage, de régularité et de non chevauchement des mailles. On peut également ajouter des termes sources dans les équation de Laplace et obtenir le système de Poisson suivant :

$$\nabla^2 \epsilon_i = P_i$$

Ce système permet de définir des lois de concentration afin d'avoir plus de mailles dans certaines régions critiques par exemple. La figure 1.5 montre un exemple d'un maillage obtenu en résolvant le système de Laplace et un obtenu en ajoutant des termes sources de façon à obtenir un maillage concentré.



(a) Système de Laplace (b) Système de Poisson

FIGURE 1.5: Exemples de maillages elliptiques obtenus en résolvant le système de Laplace et le système de Poisson.

Une autre méthode couramment utilisée consiste à diviser une géométrie complexe en plusieurs blocs plus simples. Chacun des blocs sera ensuite maillé à l'aide de l'une des méthodes déjà présentées. Ce genre de techniques produit des maillages appelés multi-blocs. Une attention particulière doit être portée aux frontières entre les blocs afin que les mailles frontières correspondent entre elles.

1.2.2 Génération de maillage non structuré

Les maillages non structurés permettent une liberté supplémentaire puisqu'on peut contrôler localement la concentration des mailles. Par contre, on doit fournir explicitement la connectivité entre les sommets.

Il existe quelques techniques pour générer des maillages non structurés. Une technique couramment utilisée est la méthode d'avance de front. En 2D, cette technique consiste d'abord à discrétiser les arêtes frontières. Puis, on ajoute graduellement des sommets et des arêtes de maillage en s'avançant à l'intérieur du front. Au fur et à mesure qu'on s'avance vers l'intérieur du domaine, des sommets sont ajoutés et des éléments sont créés. La figure 1.6 montre un exemple de progression d'un front.

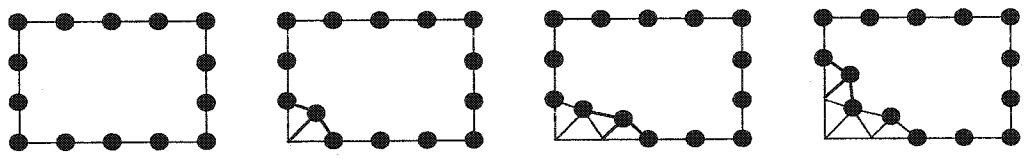


FIGURE 1.6: Progression d'un maillage construit par avance de front.

La méthode de Delaunay est également très utilisée. Cette méthode permet de générer, à partir d'un nuage de points, un maillage qui respecte une propriété intéressante. Pour un maillage de triangles, on dit qu'il respecte le critère de Delaunay si et seulement si pour chacun des triangles, le cercle circonscrit à ce triangle ne contient aucun autre sommet du maillage (outre les sommets du triangle sur lequel le cercle est construit). La méthode de Delaunay possède plusieurs avantages, mais la question à savoir comment générer le nuage de points reste entière. Une technique couramment employée consiste à générer d'abord un maillage par avance de front avant de le rendre conforme au critère de Delaunay par un retournement successif de ses arêtes. La figure 1.7 montre un exemple d'un retournement d'arête qui prend une paire d'éléments ne respectant pas ce critère et le transforme en une paire le respectant.

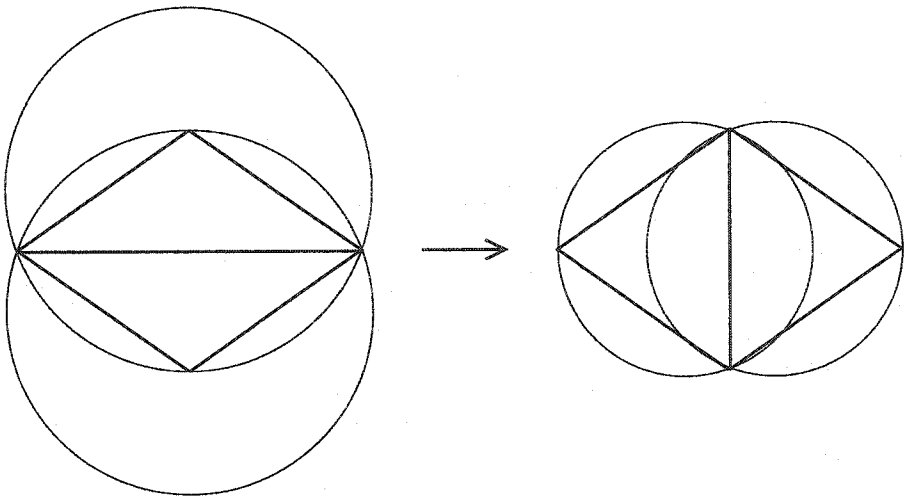


FIGURE 1.7: Exemple d'éléments invalides selon le critère de Delaunay devenus valides après un retournement d'arête.

1.2.3 Génération de maillage dans le cadre du projet

Au CERCA, le générateur de maillage développé, appelé « maille », permet d'implanter plusieurs techniques de génération de maillage distinctes. Le mailleur est générique et permet aux utilisateurs de développer leur propres algorithmes de génération de maillage (Labbé *et al.* (2001)). Plusieurs stratégies sont présentement implantées tant en 2D qu'en 3D. On peut également spécifier des lois de concentration qui permettent, dans une certaine mesure, de contrôler la concentration locale du maillage.

1.3 Adaptation de maillage

Une fois le maillage généré, si on se reporte à la figure 1.1, représentant le schéma de résolution d'une simulation numérique, l'étape suivante consiste à trouver la solution au problème à l'aide d'un résoluteur. Une fois la solution obtenue, on pourrait croire que le travail est terminé. Cependant, on sait que tout processus de résolution numérique entraîne une certaine erreur. Par exemple, il est clair que la qualité du maillage utilisé

influencera grandement la qualité de la solution. Comment faire alors pour générer un maillage satisfaisant la solution qui est à prime abord inconnue ? Ce problème peut être résolu grâce à l'adaptation de maillage. L'adaptation de maillage consiste, une fois la solution calculée, à estimer l'erreur sur cette solution et à adapter le maillage en fonction de cette erreur de façon à avoir un maillage de meilleure qualité. Ce maillage, mieux adapté à la solution, permettra au résoluteur de calculer une deuxième solution qui sera normalement plus précise que la première. Il s'agit d'un processus itératif qui peut être répété jusqu'à ce qu'on atteigne le niveau de convergence désiré.

Ainsi, le processus d'adaptation de maillage comporte deux principales étapes. La première est l'estimation d'erreur et la construction d'une métrique associée à cette erreur. L'idée est de construire une carte de taille qui guidera le remailleur chargé d'adapter le maillage. La carte de taille indique dans quelles zones le maillage doit être raffiné et dans quelles zones il pourrait être plus grossier. À partir de l'estimateur d'erreur, on construit la carte de taille en définissant une métrique sur chaque sommet du maillage de fond. Le maillage de fond correspond au maillage de référence sur lequel la carte de taille est définie. Ce pourrait être le maillage initial avant l'adaptation par exemple.

La carte de taille est représentée par une métrique. La métrique correspond à une transformation de l'espace. Cette métrique est exprimée sous forme d'une matrice (2x2 en 2D et 3x3 en 3D) symétrique définie positive. Pour chaque sommet S du maillage, on définit la matrice $\mathcal{M}(S)$ suivante :

$$\mathcal{M}(S) = \begin{bmatrix} a & b \\ b & c \end{bmatrix} \quad \text{où} \quad \begin{cases} a > 0, \quad b > 0, \\ ac - b^2 > 0 \end{cases}$$

La deuxième étape de l'adaptation consiste à adapter le maillage à partir de la métrique préalablement définie. Pour ce faire, on redéfinit le calcul de longueur dans l'espace

transformé de la métrique. Ainsi, la longueur du vecteur $\|\overrightarrow{u}\|$, normalement donnée par $\|\overrightarrow{u}\| = \sqrt{\overrightarrow{u} \cdot \overrightarrow{u}}$ dans l'espace physique habituel, devient, dans l'espace de la métrique, $\|\overrightarrow{u}\|_{\mathcal{M}} = \sqrt{\overrightarrow{u}^t \mathcal{M} \overrightarrow{u}}$. La métrique définit la taille cible du maillage. Cette définition de la longueur permet de calculer la longueur des arêtes dans l'espace métrique. Le travail du remailleur consiste alors à modifier le maillage de façon à ce que les arêtes de maillage y soient conformes. On cherche donc à ce que les arêtes soient de taille identique dans cet espace (typiquement, de longueur unitaire) et qu'elles soient orientées correctement. Ces différents concepts liés aux maillages et à l'adaptation sont couverts de façon détaillée dans les livres de Frey et George (1999) et de George (2001).

La figure 1.8 présente le schéma global du processus d'adaptation de maillage. On y voit les différentes étapes de l'adaptation. Les sections suivantes discutent des différentes techniques d'adaptation de maillage en présentant les grandes lignes de chacune d'entre elles.

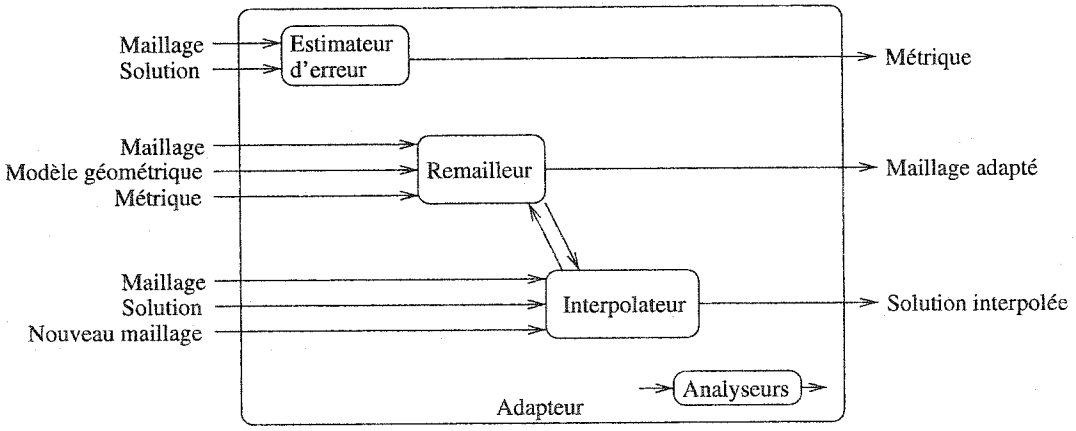


FIGURE 1.8: Schéma global du processus d'adaptation de maillage

1.3.1 Régénération du maillage

Une technique parfois utilisée pour l'adaptation de maillage est la reconstruction complète du maillage. Dans ce cas, plutôt que de raffiner certaines parties du maillage ou de déplacer des sommets, on génère un nouveau maillage en se servant de la carte de taille obtenue par estimation d'erreur comme guide. Il s'agit simplement de prendre la technique de génération de maillage appropriée. On doit pouvoir, avec la technique choisie, spécifier la densité des noeuds dans les différentes régions du maillage afin d'obtenir un maillage respectant la carte de taille.

1.3.2 Déplacement de sommets

Contrairement à toutes les autres méthodes d'adaptation de maillage, le déplacement de sommets, également appelé lissage du maillage, ne modifie pas la connectivité du maillage. Pour le cas d'un maillage structuré, si on désire garder la propriété structurée du maillage (connaissance implicite des voisins), le déplacement de sommets est la seule opération permise. La figure 1.9 montre un exemple où un sommet est déplacé. Le sommet S1 est déplacé au centre géométrique de ses voisins. On peut voir que la structure du maillage n'est pas modifiée et que les quatre éléments du maillage avant la modification sont toujours définis par les même sommets après le déplacement.

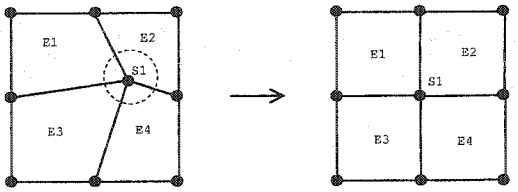


FIGURE 1.9: Exemple de déplacement d'un sommet.

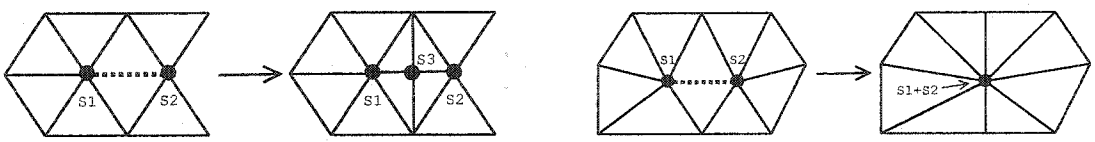
Pour déterminer la formule du déplacement, il y a plusieurs possibilités. On peut, comme dans la figure 1.9, déplacer chaque sommet au centre géométrique de ses voisins. Toute-

fois, généralement, le but recherché est de respecter la carte de taille. Dans cette optique, la formule utilisée dans **OO**$\mathcal{RT}$[1], le logiciel d'adaptation développé CERCA, est de déplacer les sommets au centre de leurs voisins *dans l'espace de la métrique* (voir Sirois *et al.* (2002)).

1.3.3 Raffinement et déraffinement

Le raffinement et le déraffinement d'arêtes sont deux opérations qui permettent de raffiner un maillage dans certaines régions et de la déraffiner dans d'autres. Cela se fait à l'aide de la carte de taille qui donne les endroits où l'erreur est forte et où il faut un maillage plus fin ainsi que les endroits où l'erreur est faible et où un maillage plus grossier ferait l'affaire.

La figure 1.10 montre un exemple de raffinement et un exemple de déraffinement. Dans le cas du raffinement (figure 1.10(a)), l'arête en pointillés a été sélectionnée pour être raffinée. Un sommet est donc ajouté en son milieu et de nouvelles arêtes sont créées pour former quatre nouveaux éléments. Dans le cas du déraffinement (figure 1.10(b)), les deux sommets de l'arête sélectionnée sont fusionnés pour devenir un seul sommet. Dans ce cas, la position du nouveau sommet a été choisie comme étant la position milieu de l'arête déraffinée, mais il s'agit d'un choix arbitraire.



(a) Raffinement (b) Déraffinement

FIGURE 1.10: Exemples de raffinement et de déraffinement d'arêtes en 2D.

[1]http ://www.cerca.umontreal.ca/oort/

1.3.4 Retournement d'arêtes et de faces

On retrouve deux types de retournement. En 2D, seul le retournement d'arêtes est permis. Toutefois, en 3D, on peut, en plus du retournement d'arêtes, faire du retournement de faces. Le principe de retournement d'arêtes en 2D est fort simple. La figure 1.7 en montre un exemple. L'arête sélectionnée est tout simplement connectée à de nouveaux sommets. Le principe du retournement en 3D est sensiblement le même.

1.4 Partitionnement

Maintenant qu'une brève revue des différents concepts touchant la génération et l'adaptation de maillage a été présentée, on peut passer au vif du sujet, soit la conception d'algorithmes parallèles pour l'adaptation. Or, pour développer une application parallèle, il est essentiel de connaître les notions de partitionnement de domaine. En effet, le but du calcul parallèle est de partager la tâche ou les données entre plusieurs processeurs. Alors, comment partage-t-on cette tâche ou cet ensemble de données ? Généralement, il s'agit de partitionner le domaine de calcul en sous-domaines qui seront distribués parmi les processeurs. Mais comment partitionne-t-on le domaine ? La question n'est pas aussi simple qu'elle ne le paraît. Deux facteurs principaux doivent être pris en compte. D'abord, il faut s'assurer d'équidistribuer la charge de travail. En effet, pour maximiser la performance du traitement parallèle, il est essentiel que tous les processeurs aient une charge de travail équivalente. De cette façon, on s'assure d'éviter que certains processeurs demeurent inoccupés pendant des périodes de temps significatives.

Dans le cas d'un maillage, on associe généralement un poids à chaque sommet. Ce poids correspond à la charge de travail reliée à ce sommet. Dans le cas où chaque sommet nécessite la même charge de travail, leur poids est tout simplement égal. Le but recherché

est donc de s'assurer que le poids total de chaque partition soit égal de façon à ne pas surcharger certains processeurs plus que d'autres. Deuxièmement, il faut également s'assurer de couper le moins d'arêtes possibles. Le nombre d'arêtes coupées (arêtes dont les deux sommets sont sur des partitions distinctes) représente une mesure de la quantité de communication qui sera nécessaire pour maintenir la cohérence des éléments aux frontières des sous-domaines. Il est évidemment souhaitable que la communication requise dans l'application parallèle soit minimale puisque les communications sont généralement beaucoup plus lentes que les calculs. Il faut donc tenter de limiter le nombre des arêtes coupées.

Avant de présenter quelques algorithmes, certaines définitions doivent être établies. Premièrement, un maillage, structuré ou non, est constitué de sommets liés entre eux par des arêtes. Généralement, pour le problème de partitionnement, on représente le maillage par un graphe de la forme $G = (V, E)$ où V représente un ensemble de sommets (*vertices*) et E un ensemble d'arêtes (*edges*). Le problème de partitionnement du maillage peut donc être vu comme un problème de partitionnement de graphe. Un partitionnement valide du graphe est un partitionnement des sommets V en partitions V_1 et V_2, tel que $V_1 \bigcup V_2 = V$ et $V_1 \bigcap V_2 = \emptyset$.

1.4.1 Algorithmes de bissection récursive

Le problème de partitionner un graphe en k partitions est un problème complexe. Il s'agit en fait d'un problème NP-complet. Plusieurs heuristiques ont été développées pour résoudre ce problème. Une méthode fréquemment utilisée est la bissection récursive. On divise d'abord le graphe en deux. Puis, chaque partition est à son tour divisée. On a donc normalement besoin de $\log_2(k)$ étapes pour diviser le graphe en k partitions. Maintenant, il reste à déterminer comment faire la division du graphe en deux. L'article de Simon (1991) présente trois techniques pour faire la bissection.

La première technique est très simple. Il s'agit d'une bissection géométrique. Cette bissection se base simplement les coordonnées des sommets du graphe. Il s'agit simplement de scinder les sommets en deux partitions en fonction d'une des coordonnées (x, y ou z). Dans un premier temps, on sélectionne la coordonnée x, y ou z selon laquelle le graphe est le plus long (en terme de nombre d'éléments). On choisit ensuite la valeur médiane sur cette coordonnée et tous les sommets sont divisés de part et d'autres de cette médiane. Une fois la division faite une première fois, on peut reprendre le processus sur chacune des partitions obtenues.

La deuxième technique de bissection est basée sur la distance entre les sommets. La distance entre deux sommets du graphe est définie comme le plus court chemin entre les deux. La longueur du chemin est déterminée par le nombre d'arêtes traversées. Pour partitionner le graphe selon cette technique, il faut d'abord trouver les deux sommets pour lesquels la distance est maximale. Ensuite, on sélectionne un de ces deux sommets et on construit une liste de sommets en ordre croissant de distance avec le sommet choisi. Finalement, on place la moitié des sommets les plus près du sommet choisi dans la première partition et l'autre moitié dans l'autre partition. Ce processus peut ensuite être ré-appliqué récursivement aux nouvelles partitions.

Un critère populaire de bissection est la méthode spectrale. Cette technique est basée sur une stratégie de bissection de graphe proposée par Pothen *et al.* (1990), elle-même basée sur la construction de la matrice de Laplace $\mathcal{L}(G)$ associée au graphe G. Cette matrice est de taille $N \times N$, où N est le nombre de sommets dans le graphe. Elle est définie comme suit :

$$\mathcal{L}(G) = \begin{bmatrix} \ddots & \vdots & \iddots \\ \cdots & l_{ij} & \cdots \\ \iddots & \vdots & \ddots \end{bmatrix} \text{ où } l_{ij} = \begin{cases} 1 \text{ si } (v_i, v_j) \in E \\ -deg(v_i) \text{ si } i = j \\ 0 \text{ sinon} \end{cases}$$

avec $deg(v_i)$, le degré du sommet v_i, c'est-à-dire le nombre d'arêtes connectées à ce sommet. Cette matrice a plusieurs propriétés intéressantes. Il s'agit d'une matrice semi-définie négative. Sa valeur propre la plus grande, notée λ_1 est nulle. De plus, si G est connecté, sa deuxième valeur propre, notée λ_2 est négative. Le vecteur propre associé à cette valeur propre λ_2, noté $\vec{u}_2$ peut alors servir de poids dans le classement des sommets en vue du partitionnement. En classant les sommets en ordre croissant de l'entrée qui leur est associée dans le vecteur $\vec{u}_2$, on peut obtenir un partitionnement par bissection. Simon (1991) a montré que pour plusieurs cas tests, cette technique génère un partitionnement de meilleure qualité comprenant moins d'arêtes coupées que les deux premières techniques de bissection récursive.

Les techniques basées sur la méthode spectrale définie précédemment sont fort populaires et il en existe plusieurs variantes. Elles sont toutes basées sur la matrice de Laplace $\mathcal{L}(G)$ et sur le vecteur propre $\vec{u}_2$ associé à la valeur propre λ_2. Toutefois, Guattery et Miller (1995) ont démontré que, pour certains types de graphes, elles peuvent produire des partitions de mauvaise qualité. De plus, selon le type de graphe, il est possible que les différentes variantes de la technique offrent des résultats différents et il devient très difficile de choisir efficacement la technique appropriée à un certain type de graphe.

Certains algorithmes ont également été développés afin de rendre la bissection récursive plus efficace. Cette approche, appelée multi-niveaux, est présentée notamment par Hendrickson et Leland (1995). Elle consiste à d'abord construire une approximation du graphe en le déraffinant. Ensuite, la bissection est appliquée sur le graphe déraffiné (plus petit) et les partitions obtenues seront projetées dans le graphe original afin d'obtenir des partitions effectives. Cette technique offre des avantages importants sur le plan de la vitesse d'exécution puisqu'il est beaucoup plus facile de partitionner un graphe plus petit.

1.4.2 Bissection multi-niveaux et METIS

La librairie METIS (voir Karypis et Kumar (1998)a) est une librairie fournissant des outils automatisés de partitionnement de graphes. L'approche préconisée par ses concepteurs est basée sur le principe de récursion multi-niveaux. Toutefois, plutôt que de procéder de façon récursive en faisant une série de bissections, le graphe plus grossier est immédiatement partitionné en k partitions. En procédant par bissection récursive, on devrait normalement procéder à $\log_2(k)$ étapes comprenant une sous étape de déraffinement, de bissection et de projection dans le graphe initial. Or, en partitionnant directement en k partitions, une seule étape sera nécessaire. Le processus de partitionnement s'effectue donc en trois phases. Ces trois phases sont présentées en détail dans Karypis et Kumar (1998)b.

Avant de décrire les trois étapes du partitionnement, la notion de poids doit être présentée. Dans METIS, comme dans plusieurs autres outils de partitionnement, il est possible de spécifier des poids aux sommets et aux arêtes du graphe. Le poids des sommets constitue une mesure de la quantité de travail à faire sur ce sommet. Il faudra donc tenter de répartir uniformément ce poids sur les différentes partitions. Ensuite, le poids des arêtes représente une mesure de dépendance entre les deux sommets liés par cette arête. En terme de traitement parallèle, cela représente la quantité de communication qui sera nécessaire entre ces deux sommets. L'algorithme de partitionnement devrait donc tenter de répartir uniformément le poids des sommets et minimiser le poids des arêtes coupées.

Ceci étant dit, la première étape de l'algorithme de METIS est la phase de déraffinement du graphe. Durant cette phase, une séquence de graphes de plus en plus petits sera construite à partir du graphe initial. Pour générer un graphe plus grossier, on fusionne des sommets ensemble. Chaque sommet du nouveau graphe aura un poids égal à la somme des poids des sommets dont il provient. Pour ce qui est des arêtes, on les construit de la façon

suivante. Soit des sommets du nouveau graphe v_1 et v_2 et les groupes de sommets d'où ils proviennent dans le graphe précédent (par fusion), respectivement V_1 et V_2. On place une arête entre v_1 et v_2 si on retrouve une arête dans le graphe initial allant d'un des sommets de V_1 à un des sommets de V_2. Le poids de cette nouvelle arête correspond au poids de l'arête dans le graphe avant déraffinement. Si plus d'une arête relie un sommet de V_1 et de V_2, on prend le poids maximal parmi les arêtes liant ces deux groupes de sommets. Cette étape de déraffinement du graphe est répétée jusqu'à ce qu'on atteigne le niveau de déraffinement voulu.

Le deuxième phase de l'algorithme est le partitionnement du graphe déraffiné obtenu lors de la première phase. Afin de procéder au partitionnement comme tel, une technique de bissection récursive est utilisée. Cette phase est décrite en détail dans Karypis et Kumar (1995). Quatre algorithmes différents sont implantés pour faire la bissection. On retrouve notamment un algorithme de bissection spectrale et de bissection géométrique, tel que présentés précédemment. À la fin de cette phase, les k partitions sont définies sur le graphe déraffiné.

La dernière phase de l'algorithme consiste à projeter les partitions obtenues dans le graphe déraffiné sur le graphe initial en suivant les même étapes que lors du déraffinement. Si on se reporte à l'étape de déraffinement, des groupes de sommets V_1 et V_2 avait été fusionnés pour devenir les sommets v_1 et v_2. On doit maintenant faire l'opération inverse. Le processus est très simple. La projection se fait en assignant les sommets du groupe V_1 à la partition du sommet v_1. À ce moment, il est possible de raffiner le partitionnement en tenant compte du poids des arêtes dans le graphe plus fin. En effet, même si le poids des arêtes coupées est minimal dans le graphe grossier, lorsqu'on projette les partitions dans le graphe plus fin, il est probable que ce poids ne soit plus minimal et que des modifications pourraient être apportées pour minimiser le poids des arêtes coupées. Un algorithme de raffinement des partitions basé sur une variante de l'algorithme

de Kernighan-Lin (Hendrickson et Leland (1995)) permet de trouver un compromis acceptable entre la répartition du poids des sommets et la minimisation du poids des arêtes coupées. Cette étape est répétée jusqu'à ce qu'on retrouve le graphe initial.

1.4.3 PARMETIS

Ainsi, METIS est une librairie qui permet de partitionner des graphes en utilisant une technique multi-niveaux. Le principal désavantage de METIS pour quelqu'un qui veut faire du calcul parallèle est que METIS fonctionne en mode séquentiel. Ainsi, toute phase de partitionnement initiale de l'application parallèle sera faite en séquentiel et une perte de performance peut en résulter. D'autant plus qu'une fois le partitionnement fait, une phase de communication sera nécessaire pour que chaque processus connaisse sa partition. C'est dans cette optique que les concepteurs de METIS ont conçu PARMETIS (voir Karypis *et al.* (2002)). PARMETIS est la version parallèle de METIS. Elle est basée sur le standard de passage de messages MPI[2].

L'avantage de PARMETIS est qu'il supporte une représentation distribuée du graphe. Le graphe n'a pas à être centralisé sur un seul processeur serveur pour ensuite être partitionné avant d'être distribué vers les autres processeurs. Ainsi, on peut maintenant partitionner des graphes dont la taille est supérieure à l'espace mémoire disponible d'un noeud puisque chaque noeud ne contient qu'une partie du graphe. De plus, dans plusieurs applications de simulation numérique, les graphes sont raffinés ou déraffinés en cours d'exécution, ce qui peut entraîner des problèmes au niveau de l'équilibre de la charge. Il devient donc essentiel de pouvoir repartitionner efficacement le graphe en parallèle afin de conserver une charge de travail équilibrée par une phase de repartitionnement peu coûteuse afin de ne pas ralentir indûment le traitement.

[2]Messages Passing Interface : http ://www.mpi-forum.org/

Les fonctionnalités fournies par ParMETIS qui sont pertinentes au contexte de l'adaptation de maillage en parallèle traité dans le présent mémoire concerne le partitionnement initial du graphe et la possibilité de le repartitionner dynamiquement. ParMETIS fourni ainsi une fonction permettant de partitionner un graphe en k partitions. Cette fonction, nommée `ParMETIS_V3_PartKway()` dans la version 3 de ParMETIS, permet à l'usager de spécifier le poids de chaque sommet et de chaque arête. À partir du communicateur MPI dans lequel évolue l'application parallèle, ParMETIS sera alors en mesure de construire les partitions et de les distribuer aux différents processeurs.

En cours d'exécution, la fonction `ParMETIS_V3_AdaptiveRepart()` permet de repartitionner le graphe. Le repartitionnement est nécessaire dans les cas où le graphe change dynamiquement (raffinement par exemple) ou simplement si le poids des sommets ou des arêtes évolue au cours du processus de calcul. `ParMETIS_V3_AdaptiveRepart()` fournit donc cette possibilité. Elle permet également de définir un ratio entre le temps requis pour faire le repartitionnement et la redistribution des sommets et la qualité des nouvelles partitions obtenues. En effet, on peut désirer des partitions de qualité moindre dans le but de diminuer le temps de repartitionnement et de redistribution des partitions. Lors d'un processus itératif heuristique, cette option peut être intéressante.

ParMETIS est donc une librairie très efficace pour faire du partitionnement de domaine et du repartitionnement dynamique. Toutefois, un problème important demeure. En effet, il est très difficile, voire impossible, avec ParMETIS de spécifier certaines contraintes sur la forme des frontières. Par exemple, l'usager pourrait vouloir spécifier une contrainte spécifiant qu'un sommet ne peut pas avoir de voisins sur plus d'une autre partition, de façon à minimiser les dépendances entre les partitions. Par exemple, dans la figure 1.11, le sommet 3, situé sur la partition P_2 n'a pour voisins que des sommets de cette même partition P_2 et d'une seule autre partition, soit la partition P_4 (sommets 6, 7 et 8). Par

contre, le sommet 2, également situé sur la partition P_2 a des voisins sur trois partitions autres que la sienne, soit les partitions P_1 (sommet 1), P_3 (sommet 5) et P_4 (sommets 6 et 7). Le chapitre 2 montre, entre autres, en quoi le fait d'avoir des partitions qui font en sorte que certains sommets ont des voisins sur plus d'une autre partition que la leur constitue un problème.

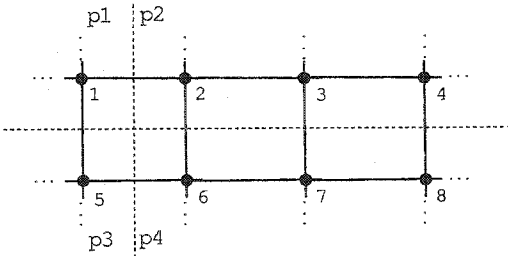


FIGURE 1.11: Exemple de sommets ayant des voisins sur plus d'une autre partition.

1.4.4 Coloriage de graphe

Soit un graphe, G, constitué d'un ensemble de sommets V et d'un ensemble d'arêtes reliant ces sommets E, tel que $G = (V, E)$. Le problème classique de coloriage de graphe consiste à associer une couleur à chacun des sommets de telle sorte que deux sommets adjacents, c'est-à-dire deux sommets partageant une arête commune, n'aient pas la même couleur. On cherche alors à ce que le nombre de couleurs utilisé pour ce faire soit minimal. Ce nombre minimal de couleurs nécessaires pour colorier le graphe G, noté $X(G)$, est appelé le nombre chromatique du graphe. Bien que le problème de coloriage soit NP-complet, plusieurs heuristiques ont été développées afin de colorier des graphes en minimisant le plus possible le nombre de couleurs utilisées. Les travaux de Karger *et al.* (1994) et de Cowen et Jesurum (1997) présentent des exemples d'algorithmes bien connus. Dans certains domaines d'application, le coloriage de graphe est appliqué sur les arêtes plutôt que sur les sommets. Le problème consiste alors à trouver le nombre minimal de couleurs à associer à chaque arête du graphe de façon à ce que

deux arêtes partageant un sommet commun n'aient pas la même couleur. Zhou et Nishizeki (2000) présentent différentes variantes de ce problème ainsi que des algorithmes pour les résoudre.

Le coloriage de graphe est une technique qui peut être utilisée au niveau du partitionnement de domaine. Sa principale application dans ce domaine est de construire des ensembles indépendants de sommets. Par exemple, l'opération de déplacement de sommets en parallèle dans le contexte de l'adaptation de maillage nécessite de maintenir la cohérence aux frontières des sous-domaines pour que le maillage reste valide en évitant les opérations conflictuelles sur des sommets dépendants l'un de l'autre situés sur des partitions différentes. Dans leur algorithme de déplacement de sommets en parallèle, Freitag *et al.* (1999) utilisent ce principe. Ils construisent des ensembles indépendants de sommets à l'aide d'un algorithme de coloriage de graphe classique de telle sorte que les sommets d'un ensemble (ayant la même couleur) sont totalement indépendants les uns des autres et peuvent être traités simultanément par des processeurs différents sans causer de problème de cohérence des frontières.

1.5 Calcul parallèle en adaptation de maillage

Relativement peu de travaux ont été faits sur l'adaptation de maillage en parallèle. Parmi les travaux réalisés, on trouve plusieurs modules d'adaptation qui sont liés à un résoluteur parallèle existant. En outre, la plupart des remailleurs parallèles ne permettent que l'opération de raffinement et parfois de déraffinement. On retrouve également des remailleurs parallèles qui sont dans les faits des générateurs de maillage en parallèle puisque l'adaptation de maillage consiste à construire un nouveau maillage. La présente section fait une brève revue de ce qui s'est fait dans ce domaine.

1.5.1 Adaptation fortement couplée au résoluteur

Dans plusieurs cas, le processus d'adaptation est fortement lié au processus de résolution. Par exemple, Sohn *et al.* (1996) ont développé un module d'adaptation de maillage et d'équilibrage de la charge de travail qui s'intègre dans un résoluteur existant. Le travail porte sur des maillages 2D de triangles et des maillages 3D de tétraèdres. Dans ce cas, le partitionnement initial est réalisé à l'aide d'une méthode de bissection récursive spectrale. Par la suite, le processus de résolution est mis en branle. Quelques itérations du résoluteur sont effectuées. Ensuite, le maillage est adapté par raffinement et déraffinement à l'aide d'informations émises par le résoluteur. Une fois cette étape franchie, on repartitionne le maillage dans le but d'équilibrer la charge. Le repartitionnement se fait sur les éléments du maillage initial (chaque nouvel élément connaît l'élément duquel il provient dans le maillage initial), ce qui permet de maintenir un temps de repartitionnement constant. Cette technique de repartitionnement est d'ailleurs fréquemment utilisée (voir Oliker et Biswas (2000) et Selwood et Berzins (1999)). Le critère utilisé pour décider si on doit repartitionner est de calculer le ratio P_{max}/P_{moy}, où P_{max} est le poids total du processeur le plus occupé et P_{moy} est le poids total moyen des processeurs. Plus ce ratio tend vers 1, plus l'équilibre de la charge est bon. Afin de calculer ce ratio, chacun des sommets du maillage maintient un poids représentant le travail associé à ce sommet.

Selwood et Berzins (1999) présentent sensiblement la même approche. Le même genre de structure de données en arbre est utilisé afin de pouvoir repartitionner le maillage efficacement. Le résoluteur marque les arêtes qui doivent être raffinées et le remailleur procédera au raffinement. Chaque processeur possède une copie de sa partition locale en plus d'un halo comprenant les sommets des partitions voisines qui sont à la frontière de la partition locale. Ceci permet au processeur de connaître la position des sommets de la partition voisine qui sont nécessaires au calcul de la position de certains sommets locaux. Il faut toutefois s'assurer que, lorsqu'un sommet est mis à jour sur un processeur,

toutes les partitions qui en possèdent une copie dans leur halo fassent une mise à jour. Pour ce faire, chaque sommet possède une liste des partitions sur lesquelles il est présent dans le halo.

Ces différentes techniques souffrent d'une lacune importante. Le module d'adaptation étant fortement couplé avec le résoluteur, il est impossible de combiner ces stratégies d'adaptation avec le résoluteur de son choix. De façon générale, on préfère avoir un module d'adaptation indépendant du résoluteur, ce qui donne plus de flexibilité. Il suffit alors de définir des interfaces standards (formats des données échangées) ou des modules de conversion de format pour être en mesure de faire les combinaisons souhaitées.

1.5.2 Mémoire distribuée versus mémoire partagée

Dans le domaine du calcul parallèle, on retrouve deux types d'architecture. Les ordinateurs à mémoire partagée possèdent plusieurs processeurs qui ont accès au même espace mémoire (adressage unique sur tous les processeurs), alors que les ordinateurs à mémoire distribuée sont constitués de plusieurs processeurs possédant leur propre mémoire et reliés entre eux par un réseau de communication. Les travaux de Oliker et Biswas (2000) présentent une comparaison de ces deux architectures dans le cadre d'une application d'adaptation de maillage. Plusieurs conclusions en sont tirées, notamment en ce qui a trait aux problèmes de synchronisation et de localité de la mémoire. En effet, on serait tenté de croire que l'architecture à mémoire partagée est beaucoup plus simple puisqu'on n'a pas besoin de se préoccuper de la distribution du maillage. Toutefois, les architectures à mémoire partagée courante ne garantissent pas un accès uniforme à la mémoire (architectures NUMA : Non Uniform Memory Access ; l'architecture NUMA-flex de SGI[3], utilisée notamment dans sa série Origin 3000[4] en constitue un exemple).

[3] www.sgi.com
[4] www.sgi.com/origin/

De plus, l'utilisation de mémoire cache locale à chaque processeur entraîne un surcoût pour maintenir la cohérence des caches. Il faut faire très attention à la localité de la mémoire et s'assurer que les données sont situées « près » du processeur leur faisant le plus fréquemment référence. Finalement, il faut aussi s'assurer que deux sommets voisins ne soient pas déplacés simultanément par deux processeurs distincts ce qui entraîne des problèmes complexes de synchronisation.

1.5.3 Adaptation par construction d'un nouveau maillage

On trouve également plusieurs travaux d'adaptation de maillage en parallèle qui consistent à reconstruire complètement le maillage à partir d'une carte de taille obtenue lors de la résolution ou par une technique d'estimation d'erreur. Souvent, des méthodes en arbres sont utilisées. Ces méthodes consistent généralement à partitionner l'espace en un certain nombre d'éléments. Ces éléments seront alors distribués aux processeurs. Chaque processeur raffine alors les éléments qui lui ont été attribués. Une attention particulière doit être apportée à la technique de génération de maillage utilisée pour que le raffinement des éléments voisins concordent.

Dans cet ordre d'idée, Pombo *et al.* (2001) ont développé un générateur de maillage en parallèle s'inscrivant dans un schéma adaptatif. Les maillages traités sont des maillages 3D non structurés et les éléments supportés sont des tétraèdres. Pour la génération du maillage, le domaine à mailler est successivement divisé en octants. Au premier niveau, on a donc 8 octants et au second, on en a 64. Ces 64 octants sont ensuite répartis parmi les processeurs. Chaque processeur raffine ensuit ses propres octants. La politique de raffinement est la même pour tous les processeurs de façon à ce que les éléments de deux octants voisins concordent.

Annamalai *et al.* (1999) ont fait des travaux similaires en 2D. Il s'agit également d'un

générateur de maillage parallèle qui s'inscrit dans un processus adaptatif. Dans ce cas, le concept d'octants est remplacé par ce qu'ils appellent des super éléments. Les super éléments constituent en fait un maillage très grossier. La carte de taille déduite par estimation d'erreur permet de connaître la densité cible du maillage sur chacun de ces super éléments, ce qui permet de les distribuer sur les différents processeurs en optimisant la répartition de la charge de travail. Chaque processeur est ensuite responsable de raffiner ses propres super éléments en fonction de la carte de taille. Les maillages supportés sont des maillages non structurés en 2D et les éléments générés sont des quadrangles.

Le principal problème de l'adaptation par construction d'un nouveau maillage est qu'on ne prend pas avantage de la connaissance du maillage initial. En fait, la génération d'un maillage adapté à la solution est un problème non linéaire. Le maillage influence la solution et la solution influence le maillage. En adaptant graduellement le maillage en fonction de la solution et en raffinant notre solution à l'aide du maillage adapté, on converge éventuellement vers une meilleure solution (et un meilleur maillage).

1.5.4 Déplacement de sommets en parallèle

Un des problèmes majeurs dans tout algorithme de déplacement des sommets en parallèle est la gestion des frontières entre les partitions. En effet, si deux sommets voisins appartiennent à des partitions distinctes situées sur des processeurs différents, rien ne peut garantir que le déplacement de ses deux sommets ne se fasse de façon sécuritaire et cela peut conduire à des éléments non conformes. Ceci s'explique par le fait que la nouvelle position d'un sommet lors du déplacement dépend de la position de ses sommets voisins. Un processeur doit donc connaître les sommets voisins de ses sommets frontières, ce qui inclut des sommets des partitions voisines. Ainsi, si les deux sommets voisins sont déplacés simultanément par deux processeurs distincts, il y a un risque de construire des éléments de mauvaise qualité ou même invalides.

Freitag *et al.* (1999) utilisent un algorithme de construction d'ensembles indépendants de sommets. L'idée est de construire des ensembles de sommets dans lesquels aucun sommet ne dépend d'un autre sommet de l'ensemble. Ainsi, chaque ensemble de sommets peut être partitionné pour être traité simultanément par plusieurs processeurs. Une mise à jour de la position des sommets sur tous les processeurs doit être faite avant de passer à l'ensemble suivant. Le problème est donc de créer des ensembles de sommets indépendants. Il s'agit en fait d'un problème typique de coloriage de graphe (coloriage des sommets). On doit attribuer une couleur à chaque sommet du graphe de telle sorte que si v_i et v_j sont deux sommets du graphe et que $\sigma(v)$ est la couleur du sommet v, on veut que $\sigma(v_i) \neq \sigma(v_j)$ si v_i et v_j sont des sommets voisins. Ainsi, les sommets de même couleur constituent des ensembles indépendants.

Le principal désavantage de cette méthode est qu'elle nécessitera une bonne quantité de communication entre le traitement de chaque ensemble indépendant puisqu'il faut mettre à jour tous les sommets de l'ensemble qui vient d'être traité sur tous les processeurs, car les sommets de deux ensembles différents peuvent être dépendants. À l'opposé, si on avait des partitions du maillage dans lesquels on avait un nombre limité de sommets frontières, on pourrait communiquer uniquement les sommets des frontières entre chaque itération.

CHAPITRE 2

ANALYSE DU PROTOTYPE D'UN REMAILLEUR PARALLÈLE

2.1 Introduction

Ce chapitre vise à présenter et à analyser un premier prototype de remailleur parallèle. Ce remailleur parallèle est basé sur une application séquentielle d'adaptation de maillage, **OO**$\mathcal{RT}$(Object Oriented Remeshing Toolkit). **OO**$\mathcal{RT}$ est une librairie de classes C++ pour l'adaptation de maillage. Cette librairie, développée au CERCA depuis plusieurs années, a produit plusieurs résultats intéressants et permet de manipuler différents types de maillages.

$\mathbb{P}$-**OO**$\mathcal{RT}$ (Parallel-**OO**$\mathcal{RT}$) est le nom utilisé pour la version parallèle de **OO**$\mathcal{RT}$. Étant donné que **OO**$\mathcal{RT}$ est toujours en développement et que des modifications sont constamment apportées dans le but d'améliorer les algorithmes d'adaptation de maillage, il était impératif que $\mathbb{P}$-**OO**$\mathcal{RT}$ hérite automatiquement des modifications apportées aux algorithmes séquentiels. C'est ce qui explique le choix de l'architecture de $\mathbb{P}$-**OO**$\mathcal{RT}$. En fait, $\mathbb{P}$-**OO**$\mathcal{RT}$ repose sur **OO**$\mathcal{RT}$ grâce au mécanisme d'héritage de classes qui permettent de redéfinir les fonctionnalités propres au traitement parallèle tout en utilisant les algorithmes d'adaptation présent dans l'application séquentielle.

Ce chapitre présente ce premier prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$. D'abord, **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$ sont présentés, puis une analyse des résultats permettra d'identifier les forces et les faiblesses du prototype.

2.2 Présentation de OO$\mathcal{RT}$

OO$\mathcal{RT}$ est une librairie orientée objets écrite en C++ offrant des fonctionnalités pour l'adaptation et l'optimisation de maillage. OO$\mathcal{RT}$ supporte les maillages structurés, non structurés et hybrides, isotropes ou anisotropes, monozone ou multizone, en 2D et en 3D. Parmi les types d'éléments supportés, on retrouve les triangles, quadrangles, hexaèdres et tétraèdres. OO$\mathcal{RT}$ utilise la librairie **PIRATE** (Labbé *et al.* (2000)), présentée lors l'introduction, pour la gestion des données géométriques, topologiques ainsi que pour les maillages et les solutions.

À partir d'un maillage existant et d'une solution calculée sur ce maillage, OO$\mathcal{RT}$ fait un calcul d'estimation d'erreur sur la solution et construit une carte de taille spécifiant la taille cible du maillage. Cette taille cible permettra ensuite d'adapter le maillage en fonction du problème à résoudre. La carte de taille est spécifiée à l'aide d'une métrique définie à chaque sommet du maillage de fond tel qu'expliqué dans le chapitre 1.

Une fois la carte de taille construite, OO$\mathcal{RT}$ procède à l'adaptation comme telle. Dans le cas des maillages structurés, seul le déplacement de sommets est permis. Chaque sommet est déplacé au centre de ses voisins dans l'espace de la métrique au cours d'un processus itératif (Sirois *et al.* (2002)). Pour les maillages non structurés, OO$\mathcal{RT}$ permet, en plus du déplacement de sommets, le raffinement d'arêtes, le déraffinement d'arêtes, le retournement d'arêtes et le retournement de faces.

2.2.1 Algorithmes de déplacement de sommets

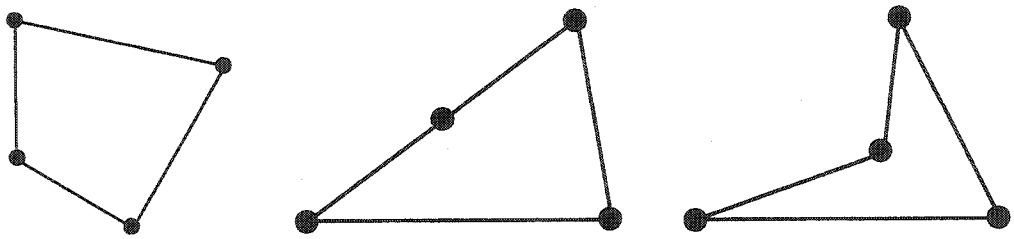
Dans un premier temps, pour des raisons qui seront présentées à la section suivante, seul le déplacement de sommets pour les maillages structurés a été implanté dans $\mathbb{P}$-OO$\mathcal{RT}$. On se concentre donc ici à décrire le fonctionnement de l'algorithme de déplacement de

sommets en séquentiel.

L'algorithme fonctionne de manière assez simple. Il s'agit de parcourir les sommets de maillage et de les déplacer un à un en fonction d'une certaine formule de déplacement. C'est un processus itératif. On refait une nouvelle itération sur tous les sommets jusqu'à ce qu'on ait atteint le niveau de convergence voulu ou que le nombre maximal d'itérations ait été atteint.

La formule de déplacement d'un sommet consiste à déplacer le sommet au centre métrique de ses voisins. Pour ce faire, on utilise la définition de la longueur dans la métrique, telle que présentée au chapitre 1 (soit en utilisant la norme $\|\overrightarrow{u}\|_{\mathcal{M}} = \sqrt{\overrightarrow{u}^t \mathcal{M} \overrightarrow{u}}$). Cette nouvelle définition de la longueur permet de calculer la nouvelle position du sommet de telle sorte qu'il se trouve à égale distance de ses voisins dans l'espace métrique. L'étape de déplacement du sommet commence donc par le calcul de sa nouvelle position. Toutefois, avant de réellement déplacer le sommet à cette nouvelle position, une vérification est faite afin de s'assurer que les éléments affectés par ce déplacement restent valides. Si la nouvelle position est invalide, elle est rejetée et le sommet n'est pas déplacé. La figure 2.1 montre trois quadrangles constitués chacun de quatre sommets et de quatre arêtes. Le quadrangle de la figure 2.1(a) est valide puisque tous les angles intérieurs sont inférieurs à 180 degrés. Toutefois, les deux derniers quadrangles sont invalides puisque l'un possède un angle plat (figure 2.1(b)) et l'autre un angle supérieur à 180 degrés (figure 2.1(c)). Ce type d'éléments, présentant une concavité, est généralement rejeté par les résoluteurs et est donc indésirable.

De par la nature de l'algorithme lui-même, il va sans dire qu'étant donné un maillage, lorsqu'une itération de déplacement est faite, il n'y a pas de solution (nouvelle position des sommets) unique. En effet, puisque la position des sommets dépend de ses voisins et que chaque sommet est déplacé à tour de rôle, la solution finale dépend de l'ordre de parcours des sommets. Toutefois, après un certain nombre d'itérations, on devrait



(a) Valide (b) Invalide (Angle plat) (c) Invalide (Angle $> 180^0$)

FIGURE 2.1: Exemples d'éléments de maillage valides et invalides en 2D.

normalement converger vers une solution optimale où les sommets sont en équilibre les uns par rapport aux autres.

Le critère de convergence utilisé est basé sur le déplacement maximal des sommets. Si le sommet s'étant déplacé le plus au cours de l'itération n a une valeur de déplacement inférieure à un certain seuil spécifié par l'usager, on considère le maillage convergé et l'opération de déplacement des sommets est interrompue. Cette valeur de déplacement est calculée dans l'espace de la métrique de façon à tenir compte de la densité locale du maillage dans le calcul du déplacement maximal. En effet, un sommet pourrait avoir un déplacement euclidien plus petit qu'un autre, mais si ce sommet est situé dans une région du maillage plus dense, ce déplacement plus faible en euclidien pourrait s'avérer plus important à l'échelle du maillage (dans la métrique). Il est à noter que l'usager peut également spécifier un nombre maximal d'itérations au delà duquel le processus de déplacement des sommets est interrompu, même si le critère de convergence n'est pas atteint.

Une optimisation supplémentaire apportée à l'algorithme consiste à procéder par plusieurs étapes au niveau du critère d'arrêt. On fixe un premier critère d'arrêt local très grossier et au cours d'une itération $k+1$, on ne déplace que les sommets dont le déplacement durant l'itération k était supérieur à ce critère. Quand tous les sommets ont atteint ce critère local, celui-ci est raffiné et on déplace à nouveaux tous les sommets. Par la suite, seuls ceux ayant un déplacement supérieur au nouveau critère local seront

déplacés jusqu'au prochain raffinement du critère d'arrêt local. On procède ainsi jusqu'à ce que le critère d'arrêt local corresponde au critère d'arrêt global spécifié par l'usager. Typiquement, cette version de l'algorithme de déplacement de sommets est de deux à quatre fois plus rapide que la version qui consiste à déplacer systématiquement tous les sommets à chaque itération. Toutefois, ceci entraînera certaines conséquences importantes dans $\mathbb{P}$-**OO**$\mathcal{RT}$.

2.3 Présentation du prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$

Tel que spécifié précédemment, $\mathbb{P}$-**OO**$\mathcal{RT}$ a été conçu de façon à réutiliser les algorithmes séquentiels présents dans **OO**$\mathcal{RT}$. L'objectif de cette architecture est d'hériter automatiquement de toutes modifications faites aux algorithmes séquentiels. Les sous sections suivantes décrivent le fonctionnement et l'architecture de $\mathbb{P}$-**OO**$\mathcal{RT}$ en détail.

2.3.1 Choix de l'algorithme à paralléliser

Dans un premier temps, il faut souligner que $\mathbb{P}$-**OO**$\mathcal{RT}$ ne supporte que les maillages structurés. Ainsi, seul l'algorithme de déplacement de sommets est parallélisé. Ce choix sera maintenant expliqué. D'abord, les autres algorithmes d'adaptation (retournement, raffinement, déraffinement) sont d'un ordre de complexité beaucoup moindre que le déplacement. Ceci vient du fait que le déplacement de sommets est un processus continu et itératif. La position d'un sommet influence tous les autres sommets dans le processus. Le déplacement de sommets adapte donc le maillage de façon globale. Les autres techniques ont, pour leur part, une influence locale sur le maillage. La décision de raffiner ou non une arête a une influence limitée aux éléments locaux situés près de cette arête. C'est pourquoi il est plus long de faire le déplacement de sommets que les autres algorithmes. Il y a donc plus à gagner à paralléliser le déplacement de sommets que

toutes autres opérations. En outre, le traitement de maillages structurés est beaucoup plus simple puisque la connectivité du maillage ne change pas en cours de calcul. Il semblait alors approprié de commencer par les maillages structurés puisque ceux-ci sont plus faciles à traiter en parallèle étant donné une topologie constante. De plus, la tâche de partitionnement et d'identification des sommets et éléments entre les processeurs est grandement facilitée par cette topologie uniforme. Pour ces différentes raisons, le prototype initial du remailleur ne supporte que les maillages structurés, tout en sachant qu'on pourrait éventuellement appliquer le déplacement de sommets en parallèle également pour des maillages non structurés.

2.3.2 Architecture

L'architecture de base de **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$est basée sur la relation d'héritage de la figure 2.2. La classe de base `REMAILLEUR_3D_GENERIQUE` est une classe abstraite qui définit l'interface de base d'un remailleur 3D et implante certaines fonctions communes aux remailleurs parallèle et séquentiel. Les classes `REMAILLEUR_3D_SEQUENTIEL` et `REMAILLEUR_3D_PARALLELE` redéfinissent certaines des fonctions propres au traitement séquentiel et au traitement parallèle.

Les principales données de maillages de **OO**$\mathcal{RT}$ sont situées dans trois listes : une liste de sommets, une liste d'arêtes et une liste d'éléments (uniquement les sommets sont présentés dans la figure 2.2, mais le principe est le même pour les arêtes et les éléments.). **OO**$\mathcal{RT}$ possède des itérateurs sur ces listes. Plutôt que de construire de nouveaux itérateurs à chaque fois que l'on doit parcourir les listes, **OO**$\mathcal{RT}$ construit et maintient un pointeur à des itérateurs qui sont construits une seule fois. Ceci permet à $\mathbb{P}$-**OO**$\mathcal{RT}$ d'utiliser les même algorithmes que **OO**$\mathcal{RT}$ simplement en redéfinissant les itérateurs de **OO**$\mathcal{RT}$ pour les faire pointer sur les sommets de la partition locale. $\mathbb{P}$-**OO**$\mathcal{RT}$ maintient ainsi une liste des sommets de la partition locale, un itérateur sur

ces sommets et un itérateur sur les sommets de tout le maillage. Ceci lui permet de modifier à volonté l'itérateur défini dans **OO**$\mathcal{RT}$ pour traiter l'ensemble des sommets ou seulement les sommets de la partition locale.

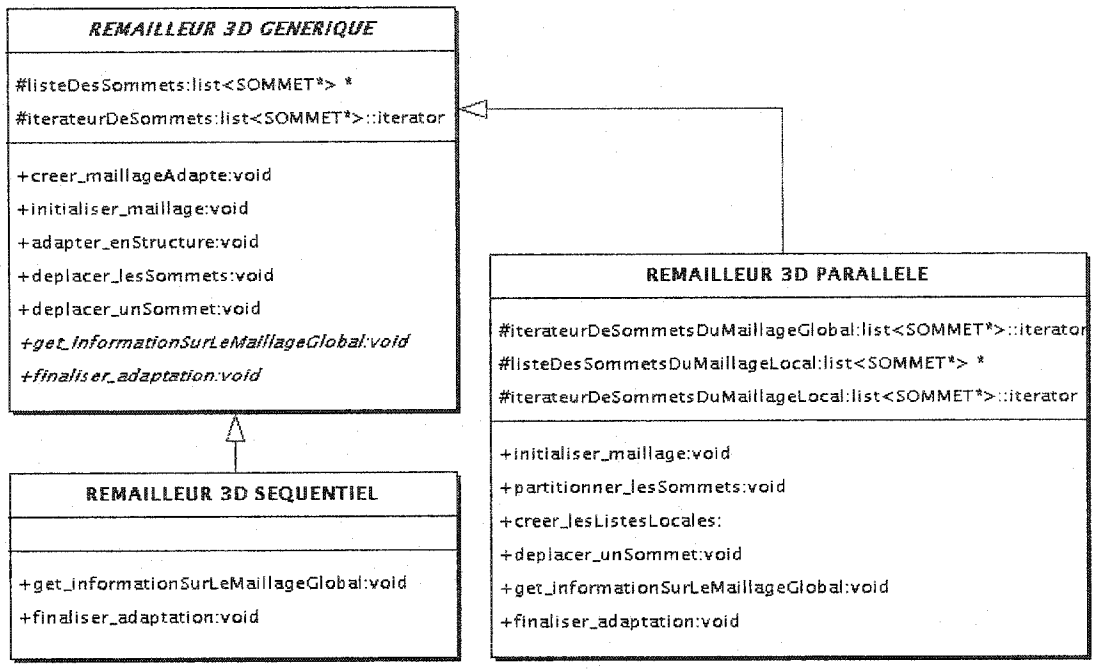


FIGURE 2.2: Diagramme de classes de base de **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$.

La fonction d'interface principale du remailleur est la fonction `creer_maillageAdapte()`. C'est cette fonction qui lance le processus d'adaptation du maillage. Ensuite, dans le cas de maillages structurés, la fonction `adapter_enStructure()` est appelée. Ces fonctions sont définies une fois pour toutes dans le `REMAILLEUR_3D_GENERIQUE`. Les fonctions `initialiser_maillage()` et `deplacer_unSommet()` sont des fonctions virtuelles qui seront redéfinies dans le `REMAILLEUR_3D_PARALLELE`. Les fonctions `get_informationSurLeMaillageGlobal()` et `finaliser_adaptation()` sont des fonctions virtuelles pures et seront redéfinies dans les remailleurs concrets. Finalement, la fonction `deplacer_lesSommets()` constitue le coeur du processus de déplacement de sommets. Il s'agit d'une méthode « template »

tel que spécifié dans le patron de conception « template method » (voir Gamma *et al.* (1995)). C'est une fonction qui est définie dans le remailleur de base, mais qui appelle des fonctions virtuelles dont le comportement diffère selon qu'on est dans la classe `REMAILLEUR_3D_SEQUENTIEL` ou `REMAILLEUR_3D_PARALLELE`.

Séquence des événements dans OO$\mathcal{RT}$

La figure 2.3, montre la séquence des appels aux fonctions principales participant au déplacement des sommets en structuré. En premier lieu, la fonction `creer_maillageAdapte()` appelle la fonction `initialiser_maillage()`. Cette fonction construit les sommets, arêtes et éléments du maillage à partir des données lues dans le fichier d'entrée et construit les listes correspondantes. Par la suite, la fonction `adapter_enStructure()` est appelée. Cette fonction appelle ensuite la fonction `deplacer_lesSommets()`. Il s'agit de la fonction centrale pour le déplacement de sommets. Cette fonction est principalement constituée de deux boucles imbriquées. La première boucle sur les itérations de déplacement et la seconde sur les sommets du maillage. Chaque sommet est alors déplacé à l'aide de la fonction `deplacer_unSommet()`. À la fin d'une itération, la fonction `get_informationSurLeMaillageGlobal()` est appelée. Dans OO$\mathcal{RT}$, cette fonction ne fait que copier les statistiques de déplacement de la partition unique dans des variables contenant les statistiques de déplacement globales à tout le maillage. Comme OO$\mathcal{RT}$ fonctionne en séquentiel et qu'une seule partition existe, les données locales et globales seront identiques. Toutefois, il est nécessaire d'appeler `get_informationSurLeMaillageGlobal()`, car $\mathbb{P}$-OO$\mathcal{RT}$ la redéfinira. Finalement, la fonction `finaliser_adaptation()`, n'est pas utile dans la version séquentielle. Elle sera toutefois redéfinie dans $\mathbb{P}$-OO$\mathcal{RT}$.

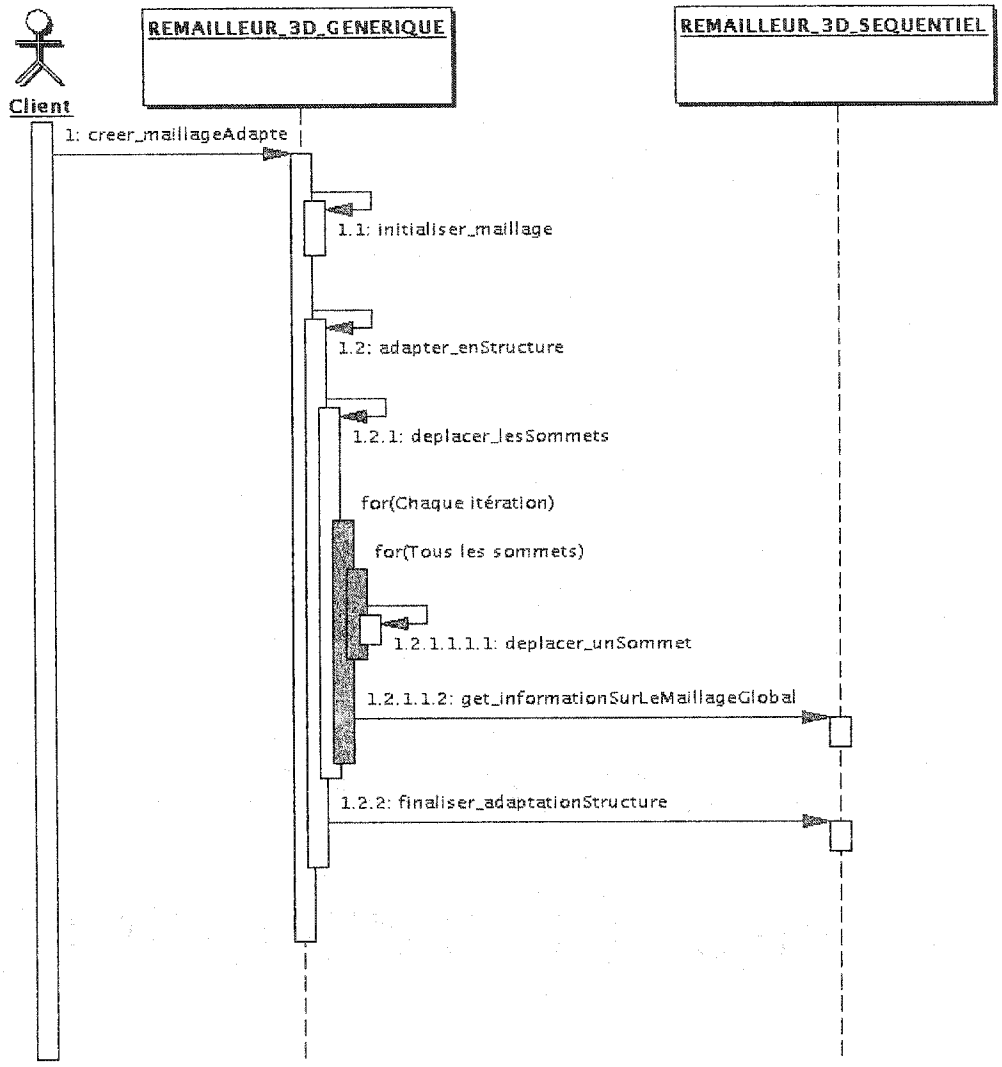


FIGURE 2.3: Diagramme de séquence du déplacement de sommets dans **OO**$\mathcal{RT}$.

Séquence des événements dans $\mathbb{P}$-**OO**$\mathcal{RT}$

La figure 2.4 montre la séquence des appels de fonction dans $\mathbb{P}$-**OO**$\mathcal{RT}$. Une première différence à noter par rapport à **OO**$\mathcal{RT}$ est qu'on utilise le `REMAILLEUR_3D_PARALLELE` plutôt que le `REMAILLEUR_3D_SEQUENTIEL`. Le `REMAILLEUR_3D_PARALLELE` redéfinit d'abord la fonction `initialiser_maillage()`. La nouvelle

version de la fonction `initialiser_maillage()`, appelle d'abord la version définie dans le `REMAILLEUR_3D_GENERIQUE`, afin de construire le maillage normalement. À la fin de ce processus chaque processeur possède une copie complète du maillage. Une fois ce processus terminé, les fonctions `partitionner_lesSommets()` et `creer_lesListesLocales()` sont appelées. Ces fonctions exécutent le partitionnement à l'aide de PARMETIS, tel qu'expliqué plus loin. Une fois le partitionnement fait, chaque processeur se construit une liste des sommets, arêtes et éléments locaux (situés sur sa partition) et construit des itérateurs pointant sur ces listes. Ces itérateurs remplaceront les itérateurs définis dans le `REMAILLEUR_3D_GENERIQUE`, afin que chaque processeur applique les algorithmes de déplacement sur les sommets de sa partition locale. Un index unique est également attribué aux sommets afin de les identifier facilement d'un processeur à l'autre.

Ensuite, la fonction `deplacer_unSommet()`, qui se charge d'appliquer la formule de déplacement à un sommet est redéfinie. En effet, dans IP-OO$\mathcal{RT}$, sur chaque processeur, on retrouve deux processus à poids léger. Un premier est chargé de faire le déplacement des sommets alors qu'un deuxième est chargé de faire la mise à jour des sommets qui ont été déplacés avec les autres processeurs Afin de s'assurer que le déplacement des sommets soient conformes, un système de verrous a été conçu. Avant de déplacer un sommet, le deuxième processus à poids léger doit obtenir le verrou de ce sommet. Une fois obtenu, il déplace effectivement le sommet en utilisant la fonction `deplacer_unSommet()` de la classe `REMAILLEUR_3D_GENERIQUE`.

La fonction `get_informationSurLeMaillageGlobal()` est également redéfinie dans le `REMAILLEUR_3D_PARALLELE`. Une première ronde de communication y est d'abord organisée afin de colliger les statistiques de déplacement des sommets sur tout le maillage (déplacement maximum, moyen, minimum, ...). Ceci permettra d'effectuer les calculs de convergence et d'afficher les statistiques pour l'usager. Un `MPI_All-`

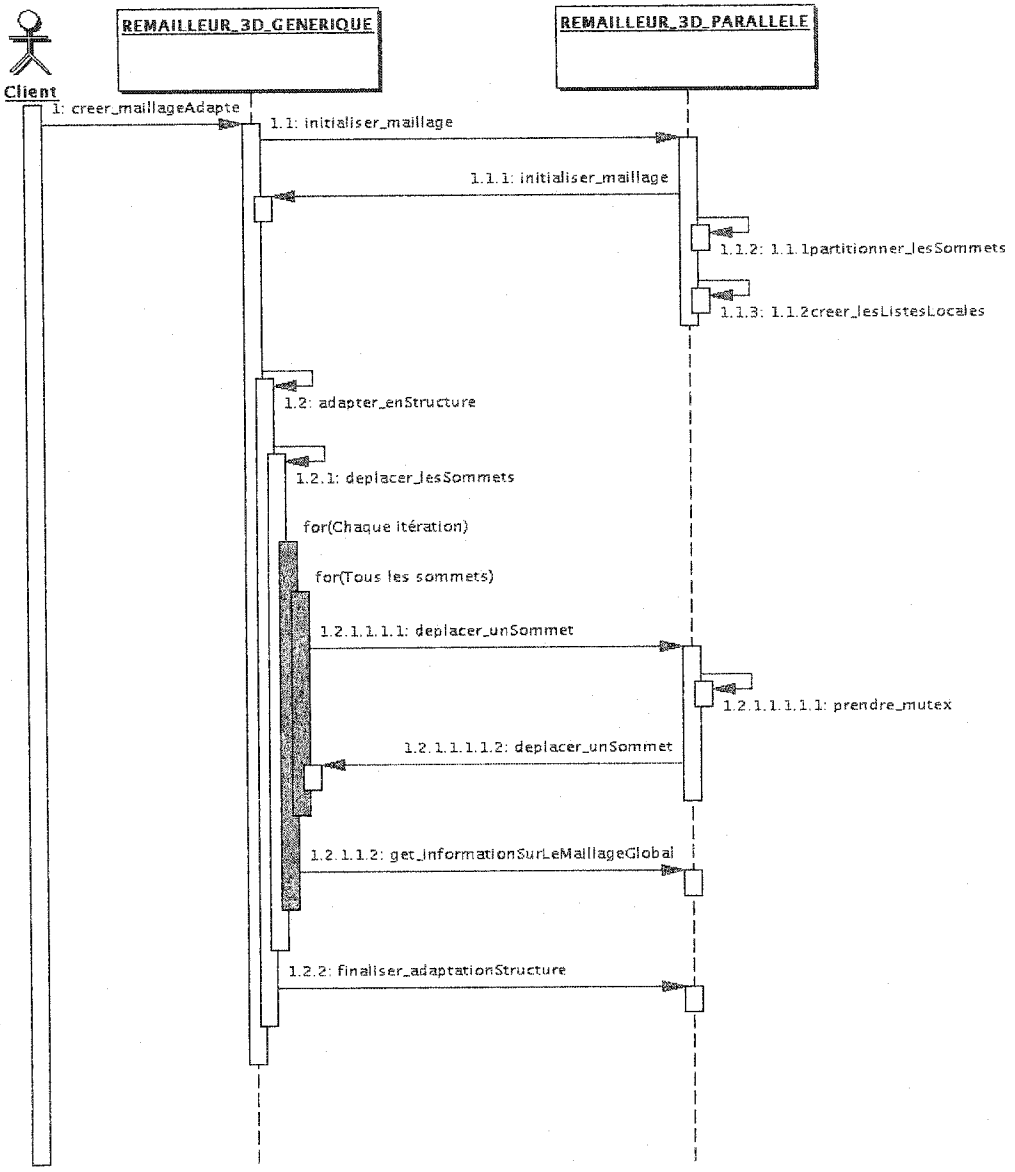


FIGURE 2.4: Diagramme de séquence du déplacement de sommets dans $\mathbb{P}$-**OO**$\mathcal{RT}$.

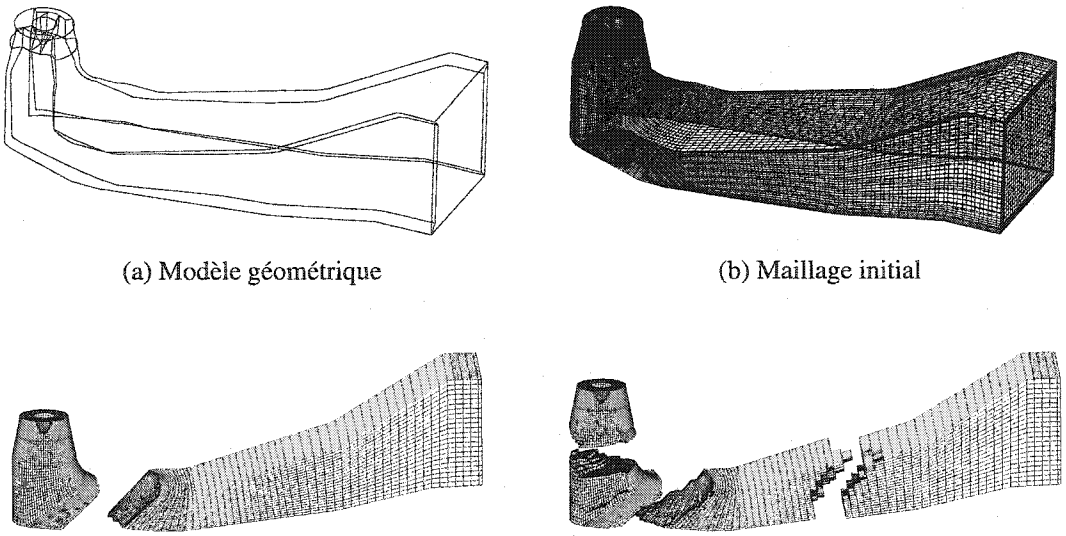
`gather()` est utilisé pour échanger les données. Une deuxième ronde de communication est ensuite organisée afin que chaque processeur communique aux autres la position des sommets qui ont été déplacés au cours de l'itération précédente. Ceci est fait en envoyant un signal au processus à poids léger responsable de la mise à jour lui indiquant de faire

la mise à jour. Ce processus utilise alors une communication impliquant des messages de longueurs variables de type « tous à tous personnalisée », soit la fonction `MPI_Alltoallv()` afin de faire la communication de tous les sommets. La transmission d'un sommet comprend son index unique et ses trois coordonnées, x, y et z. À l'aide des informations reçues, chaque processeur peut mettre à jour l'ensemble des sommets du maillage global. Ceci est très important, puisque la position des sommets dépend de leurs voisins et ceux-ci pourraient se trouver sur des partitions différentes. Une mise à jour à chaque itération est donc requise.

Finalement, la fonction `finaliser_adaptation()` permet de faire la dernière phase de communication. Cette phase consiste pour tous les processeurs de rang supérieur à 0 à envoyer la position de leurs sommets au processeur de rang 0. Celui-ci sera alors en mesure de connaître le maillage final après l'adaptation et de le sauvegarder dans le fichier de sortie.

2.3.3 Partitionnement avec PARMETIS

La phase de partitionnement du domaine s'effectue à l'aide de la librairie PARMETIS dont le fonctionnement est présenté au chapitre 1. La figure 2.5 montre un exemple de partitionnement obtenu avec PARMETIS sur le maillage d'une pièce de turbine hydraulique. On y présente le modèle géométrique qui sera maillé (figure 2.5(a)), le maillage obtenu (figure 2.5(b)) et le partitionnement obtenu avec PARMETIS en partitionnant en deux sous domaines (figure 2.5(c)) et quatre sous domaines (figure 2.5(d)). Le partitionnement a été obtenu sans spécifier de poids ni aux sommets, ni aux arêtes, ce qui correspond au cas où les poids sont identiques. Il est important de voir que les frontières entre les partitions ne sont pas parfaitement définis, en ce sens qu'un sommet peut avoir pour voisin des sommets de plus d'une autre partition. Ceci peut causer des problèmes de cohérence aux frontières tel que nous le verrons plus loin.



(a) Modèle géométrique

(b) Maillage initial

(c) Partitionnement en 2

(d) Partitionnement en 4

FIGURE 2.5: Exemple de partitionnement obtenu avec PARMETIS.

2.4 Présentation des résultats préliminaires

Cette section présente les résultats obtenus avec le prototype. D'abord, les cas tests utilisés et les environnements de test sont présentés. Ensuite, les résultats obtenus sont présentés et analysés.

2.4.1 Présentation des cas tests

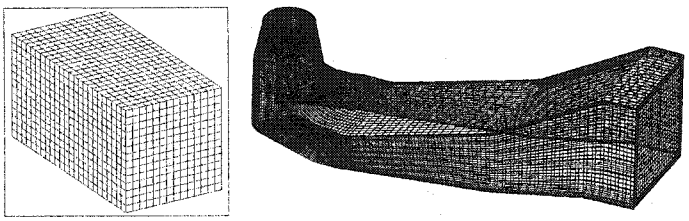
Le tableau 2.1 présente les différents cas tests utilisés pour vérifier le fonctionnement du prototype de remailleur. Les tests M1 à M3 sont en fait des variantes du même cas test, appelé Arctg. Il s'agit d'un prisme rectangulaire dont le domaine est $[0,1] \times [0,1] \times [0,2]$. Le domaine est maillé en utilisant des éléments de forme hexaédrique. Les maillages M1, M2 et M3 sont respectivement constitués de $15 \times 15 \times 25$, $30 \times 30 \times 50$ et de $60 \times 60 \times 100$ sommets. La figure 2.6(a) présente le cas test Arctg à 5625 sommets (correspond au test M1). La solution à partir de laquelle l'estimateur d'erreur générera le champs métrique

servant de guide à l'adaptation est une solution analytique de la forme $f(x, y, z) = \arctan(1000\sqrt[3]{xyz} - 250)$.

Le test M4 est basé sur le cas test Ercoftac. Il s'agit d'un aspirateur du turbine hydraulique. L'aspirateur est maillé par une technique multi-blocs. Le modèle est divisé en sept blocs et chacun de ces blocs est maillé à l'aide d'éléments hexaèdriques. La figure 2.6(b) présente le maillage initial de ce cas test. La solution numérique de l'écoulement dans ce maillage a été calculée à l'aide du logiciel commercial CFX-TASCFLOW. À partir de cette solution, l'estimateur d'erreur construit la métrique qui servira de guide à l'adaptation.

TABLEAU 2.1 Présentation des cas tests pour le prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$.

Numéro du test	Nom du cas test	Nombre de sommets	Nombre d'arêtes	Nombre d'éléments
M1	Arctg	5625	15900	4704
M2	Arctg	45000	131100	41209
M3	Arctg	360000	1064400	344619
M4	Ercoftac	127982	377829	122004



(a) Arctg (b) Ercoftac

FIGURE 2.6: Cas tests utilisés pour l'évaluation du prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$.

2.4.2 Environnements de test

Les tests sur le prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$ ont été effectués sur deux grappes de calcul Linux. La première grappe de calcul, HEYSE, est constituée de 16 noeuds. Chaque noeud

est formé d'un processeur AMD Athlon cadencé à 1.4 GHz et possédant 1 Go de mémoire. Les noeuds sont liés entre eux par un réseau Ethernet standard. HEYSE fonctionne avec le noyau 2.4.18 et le compilateur utilisé est `gcc` 2.95.3. L'implantation de MPI utilisée sur HEYSE est LAM 6.5.9[1].

La deuxième grappe, CHARYBDE, est constituée de huit noeuds. Chaque noeud est formé de quatre processeurs Pentium III Xeon cadencés à 700 MHz qui se partagent 4 Go de mémoire. Il s'agit donc d'une architecture hybride à mémoire partagée et distribuée. Les noeuds sont liés entre eux par deux réseaux distincts et il est possible d'utiliser l'un ou l'autre, au choix. Le premier est un réseau Ethernet standard. Le deuxième est un réseau Myrinet, conçu par la compagnie Myricom[2]. Il s'agit d'un réseau conçu spécialement pour les grappes de calcul qui est plus performant et qui offre de meilleures performances au niveau de la latence et de la bande passante notamment. Pour les tests effectués sur le prototype, le réseau Myrinet a été utilisé. CHARYBDE fonctionne avec le noyau 2.4.3 et le compilateur utilisé est `gcc` 2.96. L'implantation de MPI utilisée sur CHARYBDE est MPICH-GM 1.2.5..9[3], qui est l'implantation développée par Myricom spécifiquement pour les grappes de calculs utilisant le réseau Myrinet.

2.4.3 Résultats obtenus

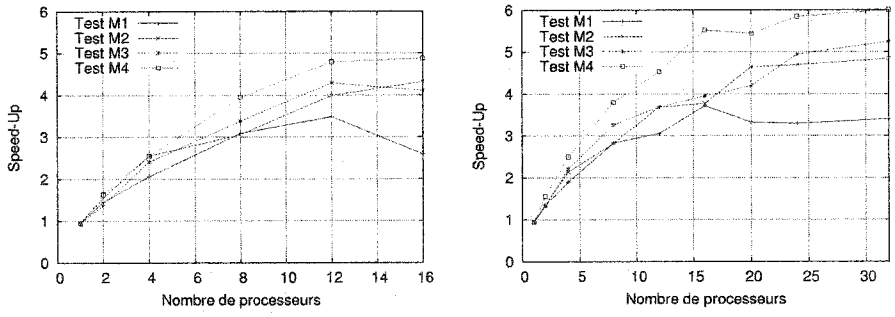
Les quatre tests M1 à M4 définis au tableau 2.1 ont été lancés sur les deux architectures décrites précédemment. Les résultats obtenus avec **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$ seront comparés dans le but de voir l'impact du traitement parallèle sur la qualité du maillage obtenu en sortie ainsi que la performance de l'application parallèle. Afin de tester la performance de $\mathbb{P}$-**OO**$\mathcal{RT}$, chacun des cas tests est lancé en faisant varier le nombre de

[1] www.lam-mpi.org/
[2] www.myri.com/
[3] www.myri.com/scs/

processeurs utilisés de façon à obtenir des courbes de speed-up. Le speed-up $S(p)$ en fonction du nombre de processeurs p est calculé de la façon suivante :

$$S(p) = \frac{t_{\text{seq}}}{t(p)}$$

Dans cette équation, t_{seq} est le temps séquentiel obtenu avec **OO**$\mathcal{RT}$ et $t(p)$ est le temps parallèle obtenu pour p processeurs. Afin d'obtenir des résultats comparables, on force **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$ à faire le même nombre d'itérations de déplacement. Ainsi, la quantité de travail faite par les deux applications sera la même, ce qui permettra de comparer les résultats correctement. La figure 2.7 montre le speed-up de $\mathbb{P}$-**OO**$\mathcal{RT}$ obtenu sur les architectures HEYSE et CHARYBDE. La figure 2.8 montre le cas test M1 dans son état initial, le résultat obtenu avec **OO**$\mathcal{RT}$ et le résultat obtenu avec $\mathbb{P}$-**OO**$\mathcal{RT}$ à quatre processeurs.

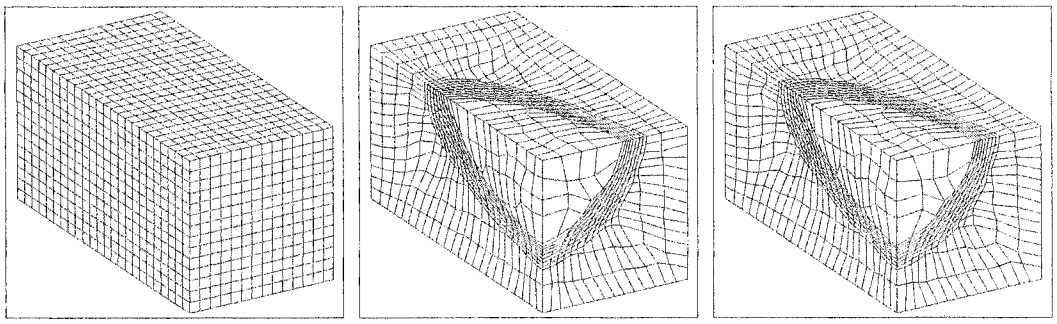


(a) HEYSE (b) CHARYBDE

FIGURE 2.7: Speed-up du prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$ sur les deux architectures

2.5 Analyse des résultats préliminaires

Les résultats préliminaires sont relativement décevants. Le premier problème majeur observé sur le prototype est la création de maillages adaptés contenant parfois des mailles



(a) Maillage initial (b) Adaptation séquentielle (c) Adaptation parallèle (4 proc.)

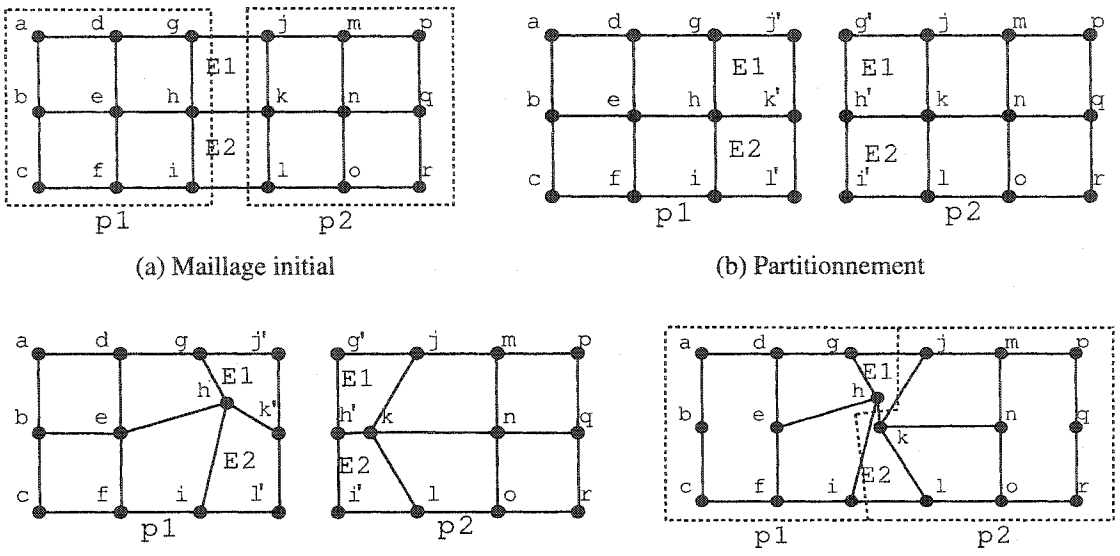
FIGURE 2.8: Comparaison des résultats obtenus lors de l'adaptation avec **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$.

non conformes. On observe également une performance très moyenne avec des speed-up assez bas qui plafonnent rapidement. Ces différents problèmes et quelques autres sont analysés en détail dans les sous sections suivantes.

2.5.1 Non conformité

Le principal problème du prototype est la génération d'éléments non conformes. Tel que mentionné précédemment, un élément non conforme est typiquement un élément présentant une concavité. Or, l'algorithme de déplacement des sommets dans **OO**$\mathcal{RT}$ vérifie, avant d'effectuer le déplacement, qu'aucun élément invalide n'est créé. Pourquoi $\mathbb{P}$-**OO**$\mathcal{RT}$, qui utilise le même algorithme de déplacement génère-t-il ce genre d'éléments ?

Le problème vient de la gestion de la cohérence des frontières. En effet, on sait que chaque processeur a une copie de tous les sommets du maillage. Toutefois, il n'en déplace qu'un sous-ensemble (sa partition) et il ne sera mis au courant de la nouvelle position des autres sommets qu'à la fin de l'itération courante. La figure 2.9 présente le schéma classique de création d'un élément non conforme.



(a) Maillage initial

(b) Partitionnement

(c) Déplacement simultané de deux sommets voisins

(d) Maillage résultant

FIGURE 2.9: Création d'éléments non conformes.

À la figure 2.9(a), on présente un maillage de quadrangles formé de 18 sommets identifiés par les lettres a à r. Lors du partitionnement, les sommets a à i seront situés sur la partition du processeur p_1 et les sommets j à r seront situés sur la partition du processeur p_2. Les éléments E_1 et E_2 sont respectivement formés des sommets (g, j, h, k) et (h, k, i, l) et possèdent tous deux des sommets sur les deux partitions distinctes. À la figure 2.9(b), on présente la situation obtenue après le partitionnement. Le processeur p_1 possède les sommets a à i et a également une copie des autres sommets, incluant les sommets voisins j', k' et l', identifiés comme tel par un symbole prime ($'$). De même, le processeur p_2 possède les sommets j à r et a une copie des autres sommets, incluant les sommets voisins g', h' et i'.

Le problème est que lorsqu'un processeur déplace un sommet, les copies de ce sommet situées sur les autres processeurs ne seront pas mises à jour avant la fin de l'itération courante. Ainsi, lorsque deux processeurs déplacent au cours de la même itération des sommets dont les positions dépendent l'une de l'autre, il y a un risque de création d'élément non conforme. On en voit un exemple à la figure 2.9(c). Le sommet h est déplacé par

le processeur p_1 et, au cour de la même itération, p_2 déplace le sommet k. Le problème est que p_1 utilise la valeur de la copie de k, k', pour calculer la nouvelle position de h et vérifier que les éléments E_1 et E_2 sont toujours valides. Il en va de même pour p_2 qui utilise la valeur de h' pour calculer la nouvelle position de k alors que cette valeur n'est plus à jour et ne correspond pas à la valeur réelle de h. Une fois que la mise à jour sera faite, on obtient, comme le montre la figure 2.9(d), deux éléments non conformes, E_1 et E_2.

En résumé, le problème vient du fait que deux processeurs distincts déplacent simultanément deux sommets dépendants l'un de l'autre. Pour régler le problème, il faudrait essayer d'éviter cette situation. Dans le cas de la figure 2.9, il faudrait par exemple qu'à une itération donnée, le processeur p_1 puisse déplacer les sommets g, h et i et que le processeur p_2 ne puisse pas déplacer les sommets j, k et l. Il suffirait d'alterner à chaque itération.

Cette technique comporte toutefois quelques désavantages. Premièrement, les sommets frontières ne seraient déplacés qu'à une itération sur deux alors que les autres le seraient à tout coup. Les sommets frontières deviendraient alors un frein à la convergence du système ralentissant ainsi la propagation de la solution globale. Il faudrait donc un plus grand nombre d'itérations pour arriver au même niveau de convergence. En outre, pour faire cela, on a supposé qu'un sommet frontière n'avait des voisins que sur une seule autre partition. Or, on sait que cela n'est pas le cas avec le partitionnement obtenu avec ParMETIS. Plusieurs sommets possèdent des voisins sur plusieurs autres partitions. Ceci s'explique en partie par le fait que ParMETIS est conçu pour le partitionnement de maillages non structurés dans lesquels ce type de frontière régulière est pratiquement impossible à obtenir. De plus, dans le cas présent, on ne parle pas uniquement de voisins par arrête commune, mais aussi de voisins par élément commun. En effet, même si deux sommets ne sont pas connectés par une arrête, s'ils appartiennent au même élément, leur

déplacement simultané par deux processeurs distincts risque de créer des éléments non conformes. La figure 2.10 montre un sommet et ses voisins par élément. On compte, pour le sommet 6, cinq sommets voisins n'étant pas sur la même partition (les sommets 3, 7, 9, 10 et 11).

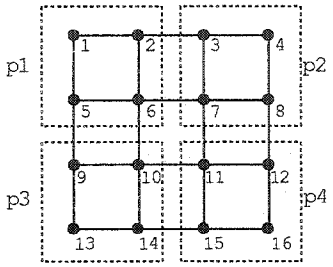


FIGURE 2.10: Un sommet et son voisinage par élément.

Une autre approche possible pourrait être de générer plus de partitions que de processeurs et de trouver ensuite des partitions indépendantes les unes des autres. Par exemple, pour 4 processeurs, on pourrait générer 16 partitions et construire quatre ensembles de partitions indépendantes. Les quatre partitions indépendantes de chaque ensemble pourrait alors être traitées simultanément par les quatre processeurs. Or, rien ne garantit qu'on sera capable de trouver un nombre de partitions par processeur permettant la construction d'ensembles de partitions indépendantes. De plus, Morin (2003) a montré que le nombre de dépendances entre les partitions obtenues avec PARMETIS était très grand et qu'il est très difficile de procéder ainsi.

Donc, pour être certain de ne pas déplacer simultanément deux sommets voisins sur deux processeurs distincts, il faudrait ne permettre qu'à un processeur à la fois de déplacer ses sommets frontières. Ceci aurait pour conséquence d'amplifier le problème de frein pour la convergence. Un test préliminaire a d'ailleurs montré qu'avec cette technique, l'écart type de la longueur des arêtes, qui devrait être le plus près possible de 0, augmente de 108%.

On en conclut que le problème est directement lié au partitionnement. En effet, si on avait une plus grande flexibilité nous permettant de contrôler la forme des frontières de façon à s'assurer qu'un sommet ne puisse pas avoir de voisins sur plus d'une autre partition, il y aurait moyen de déplacer les sommets frontières une fois sur deux et d'éviter les éléments non conformes. En outre, en étendant la notion de déplacement une fois sur deux à tous les sommets, il y aurait moyen de faire approximativement deux fois plus d'itérations, en déplaçant deux fois moins de sommets à chaque fois de façon à ce qu'aucun sommet en particulier ne soit un frein à la convergence. Ceci sera d'ailleurs examiné plus en détail dans le chapitre 3.

2.5.2 Accélération décevante

Un speed-up idéal correspond au cas où tous les processeurs travaillant en parallèle sont complètement indépendants et où la fraction du code qui reste séquentielle est quasi inexistante. Cependant, dans la vaste majorité des cas, cela ne correspond pas à la réalité. Des besoins de synchronisation entre les étapes, de partage de données et de dépendance entre des éléments de partitions différentes ajoute la nécessité d'étapes de communication et de synchronisation des processeurs qui réduisent les gains obtenus en parallèle. En outre, généralement, plus on augmente le nombre de processeurs, plus le quantité et la complexité des communications augmentent et plus la synchronisation entre les processeurs est ardue. C'est pourquoi, dans les faits, on observe rarement un speed-up idéal. La figure 2.11 montre le speed-up idéal et le speed-up typique d'une application parallèle quelconque.

Il est clair qu'on ne s'attend pas à obtenir un speed-up linéaire avec IP-OORT. Plusieurs phases de synchronisation et de communication sont nécessaires comme par exemple la communication des statistiques et la mise à jour des sommets entre chaque itération. En outre, une fraction non négligeable du code est exécutée en séquentiel. Il s'agit, entre

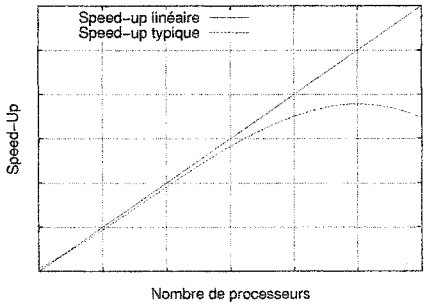


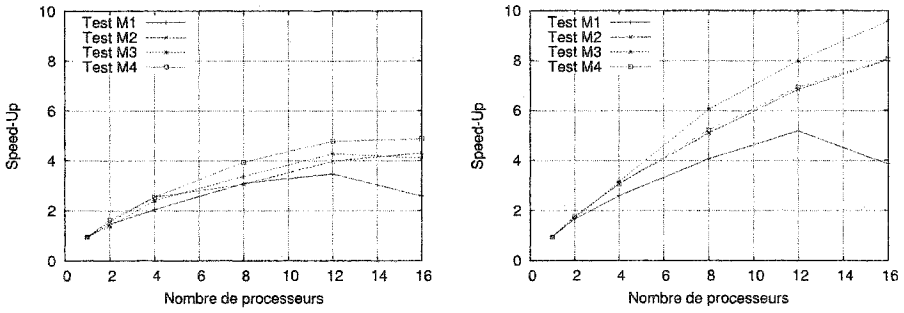
FIGURE 2.11: Speed-up linéaire et typique d'une application parallèle.

autres, de la lecture du maillage initial, de la construction de la métrique et de l'écriture du maillage final. Malgré tout cela, le speed-up obtenu, tel que présenté à la figure 2.7 est nettement insatisfaisant. Par exemple, sur l'architecture HEYSE, la performance plafonne à un speed-up inférieur à 5 sur 16 processeurs et sur CHARYBDE, elle plafonne à tout près de 6 pour 32 processeurs.

On peut également remarquer quelques particularités intrigantes. Généralement, dans une application parallèle, on s'attend à ce qu'en augmentant le volume de travail à faire, le speed-up s'améliore. En effet, plus le temps de calcul est important, plus le ratio Calcul/Communication est élevé et meilleur est le speed-up auquel on peut s'attendre. Or, dans notre cas, c'est la cas test M4 qui présente le meilleur speed-up, bien qu'il possède moins de sommets que le cas test M3 (127982 pour M4, 360000 pour M3). Pourtant, le ratio Calcul/Communication devrait être meilleur pour M3.

Après quelques tests, une explication a été trouvée à ces résultats. Il semble que le prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$ éprouve aussi un problème d'équilibre de charge. Un déséquilibre de charge se produit lorsqu'un processeur a plus de travail à faire que les autres (ou qu'il est plus lent). Dans $\mathbb{P}$-**OO**$\mathcal{RT}$, chaque partition comporte le même nombre de sommets. Cependant, on sait que **OO**$\mathcal{RT}$, dont les algorithmes de déplacement sont utilisés dans $\mathbb{P}$-**OO**$\mathcal{RT}$, ne déplacent pas systématiquement tous les sommets à chaque itération tel qu'expliqué lors de la présentation de **OO**$\mathcal{RT}$. Seuls les sommets ayant un déplacement

important seront d'abord déplacés à l'aide d'un processus de raffinement du critère d'arrêt local. Ainsi, chaque processeur n'a pas nécessairement le même nombre de sommets *à déplacer* au cours d'une itération donnée et ce, même s'il a le même nombre de sommets dans sa partition. Pour valider cette hypothèse, une série de tests a été effectuée en désactivant l'optimisation, c'est-à-dire en déplaçant systématiquement tous les sommets à chaque itération. La figure 2.12 comparent les speed-up obtenus avec et sans l'optimisation sur HEYSE.



(a) Avec l'optimisation (b) Sans l'optimisation

FIGURE 2.12: Comparaison des speed-up obtenus avec et sans l'optimisation sur HEYSE.

On voit donc qu'en désactivant l'optimisation sur le déplacement des sommets, le speed-up est nettement amélioré. Ceci confirme que $\mathbb{P}$-**OO**$\mathcal{RT}$ a un problème d'équilibre de charge puisqu'en désactivant cette optimisation, on s'assure d'un meilleur équilibre de la charge et le speed-up est meilleur. Il faut faire attention toutefois, car en désactivant cette optimisation, on améliore le speed-up, certes, mais le temps de référence séquentiel est beaucoup plus long (en moyenne de deux à quatre fois plus long). Cette optimisation permet donc de gagner du temps en séquentiel puisqu'elle conditionne le temps global réel de l'application.

On remarque également que sans l'optimisation, le cas test qui présente le meilleur résultat est le cas test M3, et non plus le cas test M4. Ainsi, la règle selon laquelle un meilleur

ratio Calcul/Communication améliore le speed-up est respectée puisque le cas test pour lequel le volume de calcul est plus élevé présente un meilleur speed-up. Il semble donc que les cas tests M1 à M3, basé sur l'Arctg, ont bénéficié davantage de l'absence de l'optimisation que le cas test M4 (Ercoftac). Ceci s'explique par le fait que l'équilibre de la charge était moins bon pour les cas tests basés sur l'Arctg que pour celui basé sur Ercoftac. En examinant le comportement du cas test Arctg, on comprend facilement pourquoi. Dans ce cas test, très peu de sommets se déplacent beaucoup et la majorité des sommets se déplacent très peu. Ainsi, les sommets demandant beaucoup de travail sont peu nombreux et tous concentrés au même endroit, ce qui surcharge les processus qui ont hérité de ces zones du maillage dans leur partition. Toutefois, quand on déplace tous les sommets, la charge est mieux répartie et le speed-up augmente substantiellement. Le speed-up d'Ercoftac augmente un peu moins tout simplement parce que la charge était déjà moins déséquilibrée. En effet, dans ce cas test, les sommets nécessitent des charges de travail plus constantes et mieux distribuées dans tout le domaine.

Ce test confirme donc un problème d'équilibre de charge dans IP-OORT. Plusieurs possibilités existent pour résoudre ce problème. La solution la plus simple serait de construire un partitionnement initial de meilleure qualité permettant de mieux répartir la charge. Cela suppose toutefois une connaissance à priori de la quantité de travail à faire sur chaque sommet, ce qui n'est pas disponible dans le cas présent. On peut également faire du repartitionnement dynamique. Après un certain nombre d'itérations, il est possible de repartitionner le maillage afin de mieux répartir la charge de travail. ParMETIS, depuis la version 3.0, fournit justement une fonction permettant de repartitionner le maillage à partir d'un partitionnement initial. Il suffit de spécifier les poids des sommets et de lancer la méthode appropriée. Toutefois, le repartitionnement a un coût, car il faut d'abord calculer le repartitionnement et, ensuite, faire migrer les éléments qui changent de partition. ParMETIS permet de définir un compromis acceptable entre la qualité des partitions obtenues du point de vue de la distribution équilibrée des poids et la

quantité d'information à échanger pour refaire le partitionnement.

2.5.3 Utilisation de la mémoire

Le dernier point du prototype à analyser concerne l'utilisation de la mémoire. Dans son état actuel, le prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$ ne répartit pas l'information parmi les différents processeurs. Au contraire, chaque processeur possède une copie complète des données ce qui peut limiter le volume d'application. Y aurait-il moyen d'être plus économe au niveau de la mémoire ? Pour répondre à cette question, il convient d'analyser plus spécifiquement les trois principales structures de données de **OO**$\mathcal{RT}$.

La première structure de données maintenues par **OO**$\mathcal{RT}$ est le modèle géométrique sur lequel est défini le maillage. Ce modèle représente les frontières du domaine sur lequel les simulations sont faites et provient généralement d'un logiciel de conception assistée par ordinateur. Le modèle géométrique est constitué d'éléments topologiques de grande dimension, les volumes, bornés par des éléments de dimension plus petite, des faces, des arêtes et des sommets. Chaque entité topologique repose sur une entité géométrique telle une surface, une courbe ou un point. Chaque élément, arête et sommet du maillage conserve une référence à l'entité topologique sur laquelle il repose. Ceci permet de s'assurer de limiter le déplacement des sommets frontières. Par exemple, un sommet situé sur une face du domaine, lorsque déplacé, doit demeurer dans cette face, de façon à ce que les frontières du modèle soient respectées. Pour l'instant, dans $\mathbb{P}$-**OO**$\mathcal{RT}$, chaque processeur possède donc une copie complète du modèle géométrique. Pour être plus efficace du point de vue de l'utilisation de la mémoire, il faudrait que lorsque le maillage est divisé en n partitions, que le modèle géométrique le soit aussi. Toutefois, la taille du modèle géométrique, bien que non négligeable, est très petite comparativement à la taille des autres structures de données de **OO**$\mathcal{RT}$. Ainsi, il n'est pas clair que le partitionnement du modèle géométrique serait très bénéfique. En outre, rien ne garantit

que le partitionnement du modèle géométrique puisse créer des sous-ensembles valides d'entités topologiques conservant leur cohérence.

La deuxième structure de données importante est le maillage de fond. Le maillage de fond est le maillage sur lequel la métrique est définie. Dans la plupart des cas, il s'agit du maillage initial avant l'adaptation. Une fonction de OO$\mathcal{RT}$ permet de localiser un point $P(x, y, z)$ dans un élément E du maillage de fond, lequel nous permet de calculer la métrique au point P en interpolant linéairement la valeur de la métrique à partir des valeurs définies aux sommets de l'élément E. Ainsi, pour partitionner le maillage de fond, il faudrait s'assurer que les partitions du maillage de fond concordent géométriquement avec les partitions du maillage adapté, de façon à ce qu'un processeur puisse localiser chacun des sommets du maillage adapté dans sa partition du maillage de fond. Les sommets situés aux frontières entre les partitions pouvant se déplacer, il s'agit là d'une tâche ardue. Il faudrait donc prévoir un recouvrement entre les partitions du maillage de fond près des frontières entre les sous domaines. Il faut donc s'assurer que lorsqu'un sommet du maillage adapté se déplace, l'élément dans lequel il se localise dans le maillage de fond soit dans la partition du processeur courant, ce qui n'est pas facile à garantir.

La dernière structure de données importantes de OO$\mathcal{RT}$ est le maillage adapté. Ce maillage est partitionné avec ParMeTiS. Le problème est qu'un processeur n'a pas uniquement besoin des sommets de sa partition pour travailler, mais également des sommets des autres partitions qui sont situés à la frontière avec sa partition puisque le déplacement d'un sommet dépend de la position de ses voisins. Présentement, la technique employée pour s'assurer que chaque processeur connaît tous les sommets nécessaires à son fonctionnement est de donner une copie complète du maillage adapté à chacun. Chaque processeur maintient ensuite une liste des sommets faisant partie de sa partition. La mémoire utilisée n'est donc pas du tout distribuée. Théoriquement, en utilisant plusieurs processeurs, on a accès à plus de mémoire. Or, IP-OO$\mathcal{RT}$ ne prend pas avantage de cela

puisque chaque processeur garde une copie de la même chose. Une technique plus efficace permettrait de répartir l'utilisation de la mémoire sur les processeurs, de sorte que chaque processeur ait uniquement une copie des sommets de sa partition et des sommets voisins de la frontière de sa partition. Cela devrait permettre de traiter des problèmes de plus grande taille. Malheureusement, le processus chargé de lire le maillage initial et d'écrire la maillage final doit quand même prévoir l'espace suffisant pour contenir tout le maillage. Ce problème pourrait être en partie maîtrisé par l'utilisation de MPI-I/O, qui permet de lire et d'écrire des fichiers en mode parallèle. Toutefois, cela impliquerait de réécrire le module faisant la lecture et l'écriture des fichiers de la librairie PIRATE[4][5] (voir Labbé *et al.* (2000)) sur laquelle reposent OO$\mathcal{RT}$ et IP-OO$\mathcal{RT}$ et rien n'était prévu en ce sens dans ce projet.

Ainsi, bien que la structure de données associée au maillage adapté puisse être distribuée de façon efficace avec une relative facilité, il en est tout autrement pour le maillage de fond et le modèle géométrique. Dans un premier temps, l'optimisation de l'utilisation de la mémoire passe donc par une distribution du maillage adapté.

2.6 Nécessité de nouveaux algorithmes de partitionnement

L'analyse faite à la section précédente démontre la nécessité de concevoir des nouveaux algorithmes de partitionnement pour IP-OO$\mathcal{RT}$. En effet, l'impossibilité de bien contrôler la forme des frontières avec PARMETIS est problématique. Cette impossibilité peut s'expliquer par le fait que PARMETIS est d'abord conçu pour les maillages non structurés. Or, l'imposition d'un grand nombre de contraintes sur un maillage non structuré pour le faire apparaître comme structuré dans METIS ou PARMETIS est une approche beaucoup trop

[4] www.polymtl.ca/grmiao/grmiao/Pir/doc/manuel/usr/usr/
[5] www.polymtl.ca/grmiao/grmiao/Pir/doc/manuel/fic/fic/

longue et hasardeuse.

En outre, un partitionnement initial mieux adapté au problème pourrait permettre de mieux répartir la charge de travail et d'améliorer le speed-up de l'application, surtout si on inclut des possibilités de repartitionnement dynamique. Quant aux problèmes d'utilisation de la mémoire, un partitionnement mieux adapté pourrait permettre aux processeurs d'identifier seulement les sommets nécessaires à son bon fonctionnement et de ne conserver que ceux-là. C'est dans ce sens qu'iront les travaux présentés au prochain chapitre.

CHAPITRE 3

NOUVEL ALGORITHME DE PARTITIONNEMENT

3.1 Introduction

L'analyse du prototype de remailleur faite au chapitre 2 a permis d'identifier quelques lacunes importantes. Premièrement, les maillages obtenus en sortie ne sont pas toujours valides. Ceci est principalement dû à la gestion des frontières. Des sommets dépendants les uns des autres situés sur des partitions distinctes sont déplacés simultanément. La deuxième lacune observée concerne le faible gain de performance obtenu en parallèle. Cette faible performance semble principalement due à un mauvais équilibre de la charge de travail. Finalement, une lacune a été observée quant à l'utilisation de la mémoire. Dans l'état courant, tous les processeurs possèdent une copie complète du maillage, ce qui n'est clairement pas optimal en ce qui concerne la quantité de mémoire utilisée.

Le nouvel algorithme devra donc :

1. Permettre de ne pas déplacer deux sommets voisins au cours d'une même itération si les deux sommets en questions se trouvent sur des partitions distinctes.
2. Éviter les problèmes de barrières de convergence. Il ne faut pas que certains sommets soient déplacés moins souvent que d'autres, car cela pourrait diminuer la vitesse de convergence. Donc, si on applique une contrainte qui est de déplacer certains sommets à une itération sur deux par exemple, la même contrainte doit être appliquée à tous les sommets. Le problème se résume donc à trouver quels sommets seront déplacés quand.
3. Améliorer la répartition de la charge de travail. Idéalement, on recherche un parti-

tionnement initial de meilleure qualité répartissant mieux la charge de travail. On cherche également un algorithme de partitionnement qui se prête bien au repartitionnement dynamique.

4. Permettre une meilleure utilisation de la mémoire. Un algorithme permettant d'identifier clairement les sommets nécessaires à chaque processeur pour fonctionner de façon à ne pas garder une copie complète du maillage serait intéressant.

Les sections suivantes traitent des différents éléments de l'algorithme. La structure de l'application ainsi que les algorithmes de partitionnement, de déplacement et de repartitionnement dynamique y sont présentés.

3.2 Structure de données

La structure de données de données utilisée pour conserver les informations sur les maillages de **OO**$\mathcal{RT}$ est fort simple. Trois classes définissent les trois entités de maillage : les sommets, les arêtes et les éléments. Les informations de connectivité sont conservées à l'intérieur même de ces classes. Par exemple, une arête connaît ses sommets et un élément connaît ses arêtes. Le `REMAILLEUR_3D_GENERIQUE` maintient une liste des sommets, une liste des arêtes et une liste des éléments.

Dans la version initiale de **OO**$\mathcal{RT}$, chacune de ces trois listes était en fait une instance d'une classe paramétrée, `LISTE<T>`. Le `REMAILLEUR_3D_GENERIQUE` fournissait des méthodes pour itérer sur chacune des trois listes. Par exemple, on pouvait y retrouver des fonctions comme `get_areteSuivante()` et `get_premiereArete()`. Le concept d'itérateur a été ensuite introduit afin de faciliter la coexistence entre **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$, permettant à $\mathbb{P}$-**OO**$\mathcal{RT}$ de redéfinir les itérateurs de **OO**$\mathcal{RT}$ de façon à appliquer les mêmes algorithmes sur un sous-ensemble de sommets (partition).

Le premier changement apporté au prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$ a donc été de changer cette structure de données de façon à la rendre plus robuste et plus facile à utiliser. Ceci étant dit, pourquoi ne pas simplement utiliser les structures de données fournies par STL ? La raison est simple. Soit une instance de liste STL servant à maintenir une liste de pointeur à des objets de la classe `SOMMET`, list<`SOMMET`*>, STL ne fournit pas de méthode permettant de supprimer un élément de la liste dans un temps de complexité $\Theta(1)$. Pour pouvoir supprimer un élément avec un temps de complexité de $\Theta(1)$, il faut absolument avoir un itérateur à accès direct à l'élément. À partir de l'élément lui-même, la complexité requise pour la suppression d'un élément sera, dans le meilleur cas, $\Theta(\log(n))$, si la liste est triée. Comme **OO**$\mathcal{RT}$ ajoute et supprime plusieurs éléments dans ses listes, il était important de garantir un temps de suppression de complexité $\Theta(1)$.

Pour ce faire, les éléments insérés dans la liste posséderont un pointeur à leurs éléments suivant et précédent, ce qui permettra à la liste doublement chaînée de facilement supprimer un élément dans un ordre de complexité constant. Les sommets, arêtes et éléments héritent donc d'une classe commune définissant un chaînon d'une liste doublement liée.

Dans $\mathbb{P}$-**OO**$\mathcal{RT}$, on utilise plutôt des vecteurs que des listes. Le vecteur global des sommets permet de retrouver facilement un sommet à partir de son index. De plus, $\mathbb{P}$-**OO**$\mathcal{RT}$ utilise également un vecteur pour gérer sa partition locale. Les vecteurs sont simplement implantés sous forme de tableaux dynamiques contenant des pointeurs à des sommets. Toutefois, pour permettre à $\mathbb{P}$-**OO**$\mathcal{RT}$ de redéfinir les itérateurs de **OO**$\mathcal{RT}$, il fallait créer une interface commune pour des itérateurs sur des listes doublement chaînées et sur des vecteurs. La figure 3.1 montre le diagramme des classes de cette structure de données. Dans le cas d'un itérateur sur une liste doublement chaînée, une attention particulière a été portée au comportement des itérateurs lors de la suppression d'un élément. La patron « observer » (voir Gamma *et al.* (1995)) a été utilisé. La liste enregistre les itérateurs qui la parcourent et elle les avertit lors de la suppression d'un élément.

Ainsi, les itérateurs peuvent s'assurer de ne pas pointer sur un élément invalide. Dans le cas des vecteurs, ceci n'est pas utile puisqu'on ne supprimera pas d'éléments de ceux-ci.

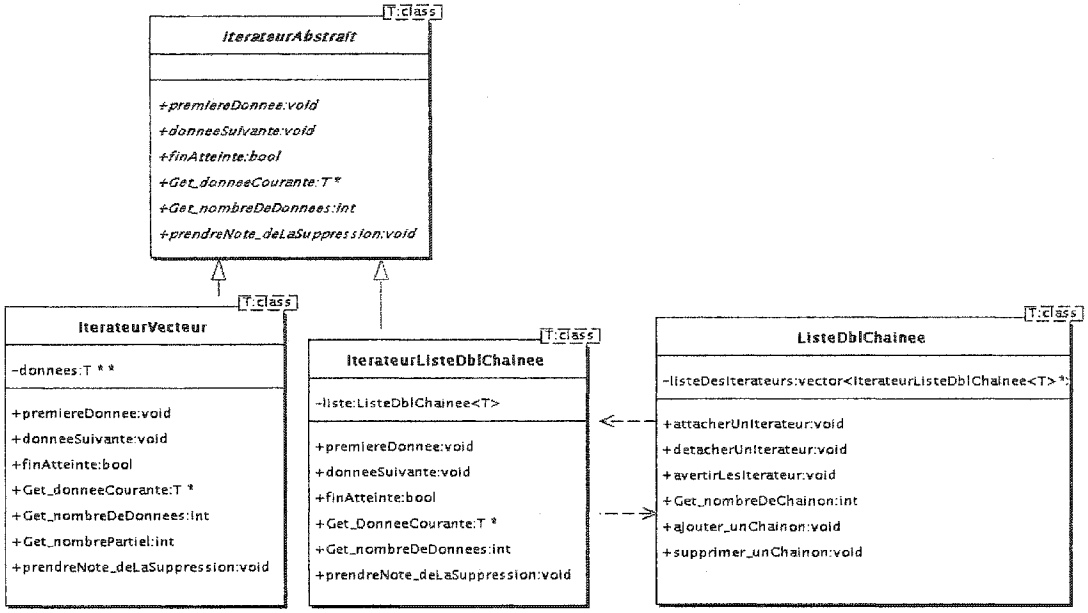


FIGURE 3.1: Diagramme de classes de la structure de données.

Les itérateurs de vecteur permettent également d'itérer sur une portion de vecteur. Il suffit à l'usager de spécifier l'index du premier et du dernier élément pointé dans le vecteur lors de la construction de l'itérateur. On définit également au moment de la construction le sens de parcours des données pointées par l'itérateur.

3.3 Algorithme de partitionnement

L'algorithme de partitionnement est encapsulé dans la classe `PARTITIONNEUR`. Le diagramme de séquence de la figure 3.2 présente les principales étapes de cet algorithme. Chacune de ces étapes sera présentée en détail dans les sous-sections suivantes.

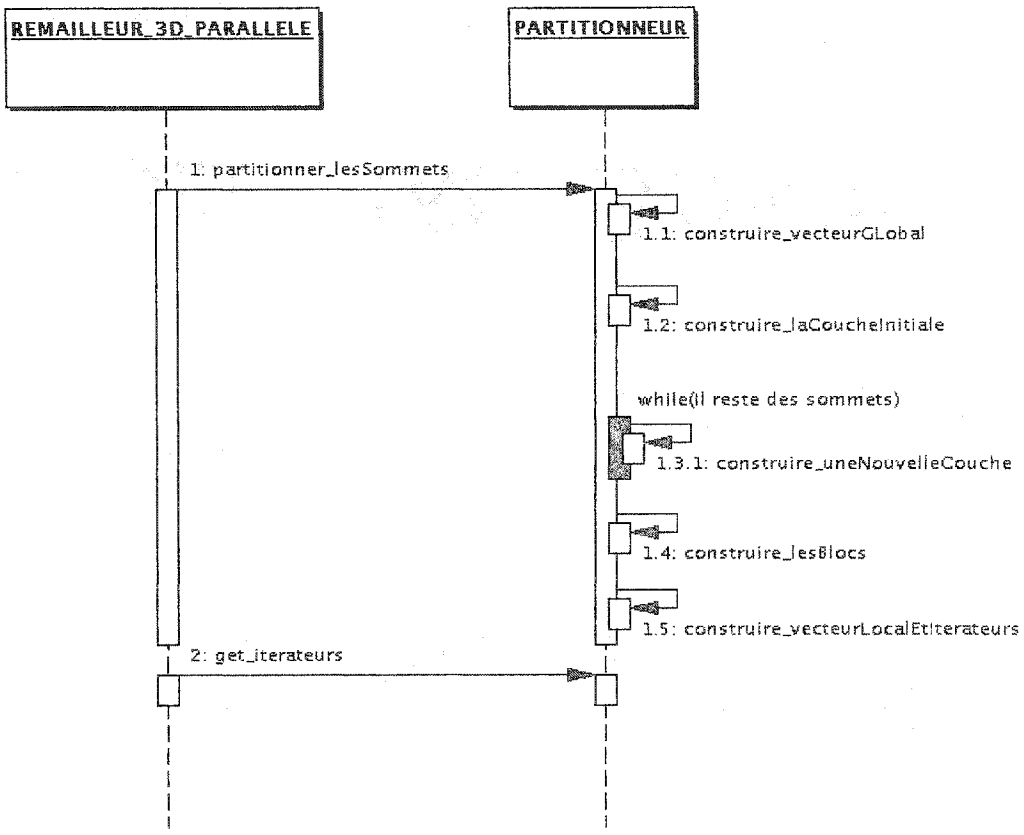


FIGURE 3.2: Diagramme de séquence du processus de partitionnement.

3.3.1 Construction du vecteur global

La première étape du processus de partitionnement (étape 1.1 du diagramme de la figure 3.2) consiste à construire un vecteur global et à attribuer un index unique aux sommets de façon à les identifier facilement à travers tous les processeurs. Chaque processeur possède une copie complète du maillage initial lue à partir du fichier d'entrée. Il suffit donc, pour chaque processeur, de parcourir la liste dans l'ordre et d'attribuer un index aux sommets allant de 0 à $n - 1$, où n est le nombre de sommets. Comme les sommets sont contenus dans le même ordre sur chaque processeur, les index attribués correspondront entre eux. À cette étape, on construit également un vecteur de pointeurs

aux sommets du maillage. Ce vecteur sera construit de telle sorte qu'à partir de l'index on puisse facilement retrouver le pointeur au sommet correspondant dans le vecteur. Ainsi, l'index attribué à un sommet correspond également à sa position dans le vecteur des sommets. Cela facilitera la recherche des sommets lors des nombreuses mises à jour.

3.3.2 Construire les couches de sommets

Une fois le vecteur global construit, le partitionnement comme tel peut commencer. L'idée de base du partitionnement repose sur une technique de coloriage de graphe. En fait, il s'agit de diviser les sommets en couches de sommets (une couche peut être associée à une couleur dans un algorithme de coloriage de graphe classique). Les couches (ou couleurs) devront respecter plusieurs contraintes. D'abord, toute couche C_i, à l'exception de la première et de la dernière, aura exactement deux couches voisines, C_{i-1} et C_{i+1}. La première et la dernière couches auront chacune une seule couche voisine. Lorsque deux couches C_x et C_y ne sont pas voisines l'une de l'autre, aucun des sommets de la couche C_x ne doit avoir pour voisin un sommet de la couche C_y. Autrement dit, un sommet situé sur une couche C_i ne peut avoir pour voisins que des sommets de la même couche (C_i) ou des deux couches voisines (C_{i-1} et C_{i+1}). Deux sommets sont considérés voisins s'ils appartiennent à un même élément de maillage.

Pour amorcer le processus, l'usager doit spécifier une face extérieure du domaine maillé. Tous les sommets de maillage situés sur cette face extérieure seront placés sur la première couche, C_0 (étape 1.2 du diagramme de la figure 3.2). Cet algorithme est présenté à la figure 3.3. D'abord, la liste de tous les éléments topologiques constituant la face de départ est construite. Ensuite, on parcoure les sommets du maillage et ceux dont l'élément topologique associé fait partie de la liste, sont ajoutés à la couche initiale.

Une fois la couche initiale construite, on passe aux couches suivantes (étape 1.3 du dia-

```
Données : etatDesSommets[nombreDeSommets]
Initialiser les données de « etatDesSommets » à −1;
Construire la liste des éléments topologiques constituant la face de départ;
pour chaque Tous les sommets de maillage faire
    si L'élément topologique sur lequel repose le sommet fait partie des éléments topologiques constituant la face de départ alors
        Ajouter le sommet à la couche initiale;
        etatDesSommets[Index du sommet courant] = 1;
    fin
fin
```

FIGURE 3.3: Algorithme de construction de la couche initiale.

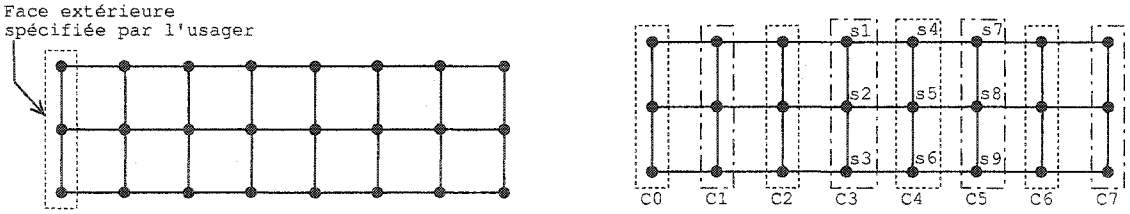
gramme de la figure 3.2). Pour construire une nouvelle couche C_{i+1}, les sommets de la couche C_i sont parcourus. Tous les sommets voisins des sommets de la couche C_i qui n'ont pas encore été assignés à une couche sont alors assignés à la nouvelle couche C_{i+1}. Afin de savoir si un sommet a déjà été assigné à une couche, on utilise un tableau contenant une entrée pour chaque sommet. Quand un sommet est placé sur une couche, on l'indique en modifiant le valeur correspondante dans le tableau (en utilisant l'index du sommet). Ce tableau est construit et initialisé au tout début de la construction de la couche initiale (figure 3.3). L'entrée de chacun des sommets dans le vecteur est initialisée à -1 pour signifier que le sommet n'a pas été assigné à une couche et lorsqu'il est assigné, on l'indique en modifiant l'entrée correspondante dans le tableau à 1. Éventuellement, il serait très facile de modifier l'algorithme pour que l'entrée correspondante à un sommet dans le tableau contienne en fait le numéro de la couche sur laquelle il a été assigné. Toutefois, dans le cas présent, cette information n'était pas utile puisqu'il ne s'agit que d'un tableau temporaire qui sera détruit aussitôt que la construction des couches sera complétée. L'algorithme de construction des couches suivantes est présenté à la figure 3.4.

La figure 3.5(a) montre une vue 2D d'un maillage sur lequel l'algorithme de division en couches est appliqué. La figure 3.5(b) montre les 8 couches obtenues, numérotées de 0

```
Données : etatDesSommets[nombreDeSommets]
Données : coucheInitiale
couchePrécédente = coucheInitiale;
tant que il reste des couches faire
    pour chaque sommet de la couchePrécédente faire
        si etatDesSommets[sommetCourant] = -1 alors
            Ajouter le sommet à la nouvelleCouche;
            etatDesSommets[sommetCourant] = 1;
        fin
    fin
    Ajouter « nouvelleCouche » à la liste des couches;
    couchePrécédente = nouvelleCouche;
fin
```

FIGURE 3.4: Algorithme de construction des couches suivantes.

à 7. On y voit que pour chaque couche, un sommet qui en fait partie ne dépend que des sommets de la même couche ou des deux couches voisines. Par exemple, le sommet s_5, situé sur la couche C_4 possède des voisins sur cette même couche (s_4 ets_6) ainsi que sur les deux couches voisines C_3 (s_1, s_2 et s_3) et C_5 (s_7, s_8 et s_9), mais sur aucune autre couche. Pour un maillage structuré mono bloc, chaque couche aura le même nombre de sommets. Toutefois, comme le montre la figure 3.6, pour un maillage multi-blocs, la taille des couches peut varier.



(a) Maillage initial

(b) Les couches obtenues

FIGURE 3.5: Vue 2D d'un maillage divisé en couches.

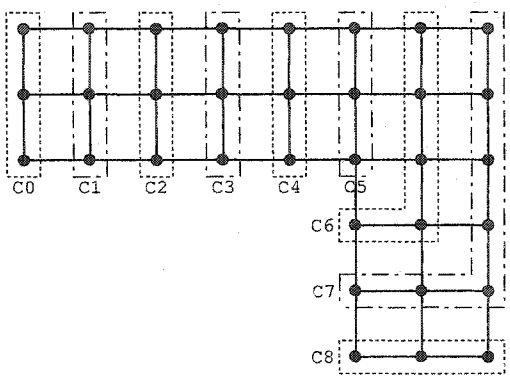


FIGURE 3.6: Maillage multi-blocs divisé en couches.

3.3.3 Construire les blocs de sommets

Une fois les couches construites, la prochaine étape est de construire les blocs (étape 1.4 du diagramme de la figure 3.2). Un bloc sera constitué de :

1. Une couche « frontière gauche ».
2. Un certain nombre n de couches internes. Ce nombre est spécifié par l'usager.
3. Une couche « frontière droite ».

Les partitions de chaque processeur sont construites en même temps que les blocs. Un bloc est alloué à un processeur et le bloc suivant sera attribué au processeur suivant. Une partition comprendra un ou plusieurs blocs, selon la taille du maillage et la taille des blocs. Quand tous les processeurs ont reçu un bloc, on recommence avec le premier processeur. On continue jusqu'à ce que toutes les couches aient été attribuées.

Au cours de ce processus de construction et d'attribution des blocs, les processeurs se créent chacun cinq listes temporaires qui leur permettront de connaître la liste de leurs sommets divisés par catégories :

1. Une liste contenant les sommets des couches « frontières gauches » de tous les blocs attribués à ce processeur.

2. Une liste contenant les sommets des couches « frontières droites » de tous les blocs attribués à ce processeur.
3. Une liste contenant les sommets des couches internes de tous les blocs attribués à ce processeur.
4. Une liste contenant les sommets des couches « frontières droite » des blocs attribués au processeur PRÉCÉDENT. Cette liste représente les sommets situés sur le processeur précédent dont la position affecte les sommets de la frontière gauche du processeur courant.
5. Une liste contenant les sommets des couches « frontières gauche » des blocs attribués au processeur SUIVANT. Cette liste représente les sommets situés sur le processeur suivant dont la position affecte les sommets de la frontière droite du processeur courant.

La figure 3.7 présente l'algorithme de construction et d'allocation des blocs. Chaque processeur parcoure la liste des couches construite à l'étape précédente. Il s'agit en fait d'une liste contenant des listes de sommets, chacune de ces listes contenant les sommets associés à une couche. L'algorithme de construction des couches a permis à chaque processeur de construire la liste des couches. On détermine donc un processeur à qui attribuer le bloc courant et chaque processeur, en fonction de son rang, sait s'il doit tenir compte du bloc courant. À toutes les fois qu'un bloc est attribué, on incrémente le numéro du processeur à qui le bloc en construction est attribué et le nouveau bloc sera ainsi attribué au prochain processeur.

Si jamais le nombre de couches n'est pas un multiple du nombre de couches par blocs, les premiers blocs auront une couche interne supplémentaire. Le nombre de couches internes à mettre dans chacun des blocs est calculé préalablement et conservé dans un vecteur.

```
numeroDeLaCouche = 0;
numeroDuProcesseur = 0;
indexDuBloc = 0;
tant que numeroDeLaCouche < nombreDeCouches faire
    si rangDuProcesseur = numeroDuProcesseur alors
        Ajouter les sommets de la couche « numeroDeLaCouche » à la liste des
        sommets de la frontière gauche du processeur;
    sinon si rangDuProcesseur = (numeroDuProcesseur-1) alors
        Ajouter les sommets de la couche « numeroDeLaCouche » à la liste des
        sommets de la frontière gauche du processeur suivant;
    Incrémenter numeroDeLaCouche;
    pour Toutes les couches internes du bloc courant faire
        si rangDuProcesseur = numeroDuProcesseur alors
            Ajouter les sommets de la couche « numeroDeLaCouche » à la liste
            des sommets internes du processeur;
        Incrémenter numeroDeLaCouche;
    fin
    si rangDuProcesseur = numeroDuProcesseur alors
        Ajouter les sommets de la couche « numeroDeLaCouche » à la liste des
        sommets de la frontière droite du processeur;
    sinon si rangDuProcesseur = (numeroDuProcesseur+1) alors
        Ajouter les sommets de la couche « numeroDeLaCouche » à la liste des
        sommets de la frontière droite du processeur précédent;
    Incrémenter numeroDeLaCouche;
    Incrémenter indexDuBloc;
    Incrémenter numeroDuProcesseur (modulo le nombre de processeurs);
fin
```

FIGURE 3.7: Algorithme de construction et d'allocation des blocs.

La figure 3.8 montre un exemple de blocs répartis sur différents processeurs. Le maillage a été divisé en 24 couches, numérotées de 0 à 23. À partir de ces couches, l'algorithme de construction et d'allocation des blocs a, pour chaque processeur, construit les différentes listes. Dans le cas présent, il y a trois processeurs qui se partagent la tâche et chaque bloc est constitué de deux couches internes. À la fin de l'attribution des couches, le processeur P_1 possède trois listes correspondant aux sommets de son domaine : une liste des sommets de sa frontière gauche (couches C_4 et C_{16}), une liste de ses sommets

internes (couches C_5, C_6, C_{17} et C_{18}) et une liste des sommets de sa frontière droite (couches C_7 et C_{19}). Il possède également deux listes supplémentaires, soit les listes des sommets voisins de ses propres frontières situés sur les processeurs suivant et précédent. Ainsi, il possède une liste des sommets des frontières droites du processeur précédent, P_0, (couche C_3 et C_{15}) et une liste des sommets des frontières gauche du processeur suivant, P_2 (couches C_8 et C_{20}).

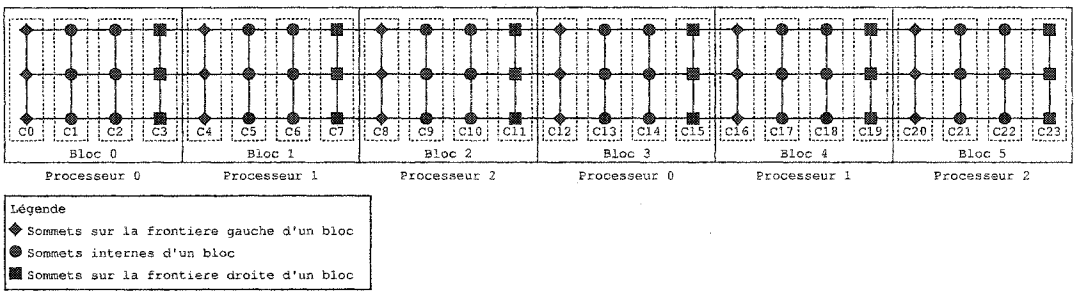


FIGURE 3.8: Répartition des blocs sur les processeurs.

Maintenant, que se passe-t-il avec les cas particuliers de la première couche (C_0 dans la figure 3.8) et de la dernière couche (C_{23} dans la figure 3.8) ? Bien que ces couches soient situées à la frontière d'un bloc, elles ne sont pas connectées à la frontière d'un autre bloc. Elles pourraient donc être considérées comme des couches internes. Toutefois, par souci d'uniformité, on fonctionne en boucle. L'algorithme considère donc que la couche C_{23} a pour voisine la couche C_0 et vice-versa.

À partir de ces cinq listes temporaires, chaque processeur se construit un vecteur contenant les différents éléments ainsi que des itérateurs sur différentes parties de ce vecteur (étape 1.5 du diagramme de la figure 3.2). Le vecteur contient dans l'ordre :

1. Les sommets du processeur précédent situés à la frontière (zone 1).
2. Les sommets de la frontière gauche (zone 2).
3. Les sommets internes (zone 3).
4. Les sommets de la frontière droite (zone 4).

5. Les sommets du processeur suivant situés à la frontière (zone 5).

On construit alors différents itérateurs pointant sur différentes parties de ce vecteur. Il y aura d'abord quatre itérateurs servant pour les mises à jour. Ces itérateurs pointent sur les zones 1, 2, 4 et 5. Ils permettront d'itérer sur les sommets à envoyer et à mettre à jour lors de la phase de synchronisation des frontières.

Deux autres itérateurs seront créés. Ces itérateurs permettront d'itérer sur les sommets à déplacer lors de chaque itération. Le premier itérera sur les sommets de la frontière gauche et sur la première moitié des sommets internes, alors que le deuxième itérera sur la deuxième moitié des sommets internes et sur les sommets de la frontière droite. Après avoir appelé la fonction `partitionner_lesSommets()`, le `REMAILLEUR_3D_PARALLELE` prend connaissance des itérateurs créés par le `PARTITIONNEUR`.

Au moment du partitionnement, aucune communication entre les différents processeurs n'est requise. Chaque processeur a lu le fichier d'entrée et construit une structure de données identique. Chaque processeur connaît son rang et le nombre de processeurs impliqués et se base sur le fait que chacun des autres processeurs possède une structure de données identique (les sommets, arêtes et éléments de maillage sont stockés dans le même ordre dans les listes) pour appliquer les algorithmes de partitionnement sans avoir à communiquer.

3.4 Algorithme de déplacement

Une fois le partitionnement terminé, le processus de déplacement des sommets peut s'amorcer. Dans OORT, deux itérateurs sont utilisés pour parcourir les sommets. À chaque itération, on alterne les itérateurs utilisés. Ceci permet de parcourir les sommets en sens inverse une fois sur deux. Cette modification permet d'améliorer la qualité du maillage en sortie.

3.4.1 Itérations paires et impaires

Dans $\mathbb{P}$-**OO**$\mathcal{RT}$, on cherche à ce que deux sommets voisins ne soient pas déplacés par deux processeurs distincts au cours de la même itération. Avec le partitionnement obtenu à la section précédente, on peut voir tout de suite que si les processeurs s'entendent pour ne déplacer qu'une frontière à la fois (gauche ou droite), on s'assure de ne pas créer d'éléments invalides au cours de l'itération. Par exemple, sur la figure 3.8, supposons qu'au cours d'une itération, les processeurs ne déplacent pas leur sommets situés sur la frontière droite. Lorsque le processeur 1 décide de déplacer un sommet de la couche C_4 (frontière gauche), il peut le faire sans problème, car la position de tous les voisins est connue. Les sommets des couches C_4 et C_5 sont évidemment connus puisqu'ils reposent sur le même processeur, alors que les sommets de la couche C_3 sont également connus, car étant sur la frontière droite du processeur précédent, ils n'ont pas été déplacé au cours de l'itération courante.

Afin de s'assurer que le maillage reste valide, il suffit d'empêcher le déplacement d'une des deux frontières de chaque processeur au cours de l'itération. Les sommets internes pourraient être déplacés au cours de toutes les itérations sans causer de problème. Toutefois, afin de s'assurer que tous les sommets du maillage sont traités à la même fréquence, on déplacera aussi ces sommets une fois sur deux.

Pour ce faire, le `REMAILLEUR_3D_PARALLELE` redéfinit les itérateurs utilisés par **OO**$\mathcal{RT}$ (dans le `REMAILLEUR_3D_GENERIQUE`) pour qu'ils pointent sur les itérateurs construits par le `PARTITIONNEUR`. Ainsi, lors des itérations impaires, les processeurs déplacent la première moitié de leurs sommets internes en plus des sommets de la frontière gauche et lors de itérations paires, les processeurs déplacent la deuxième moitié de leurs sommets internes en plus des sommets de la frontière droite. Ce procédé s'apparente à l'algorithme de tri pair impair (Kornerup (1997)) fréquemment utilisé en

parallélisme.

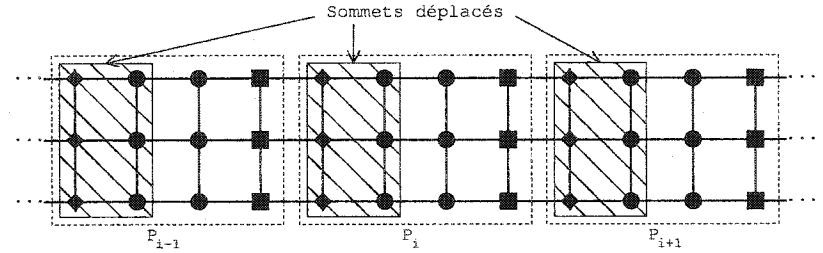
Le comportement du processus itératif de $\mathbb{P}$-OO$\mathcal{RT}$ est donc différent du comportement de celui de OO$\mathcal{RT}$ puisque les sommets ne seront pas parcourus dans le même ordre. Une attention particulière devra être portée lors de la vérification du programme (voir chapitre 4) afin de vérifier si cela affecte la qualité des maillages obtenus.

3.4.2 Synchronisation inter itération

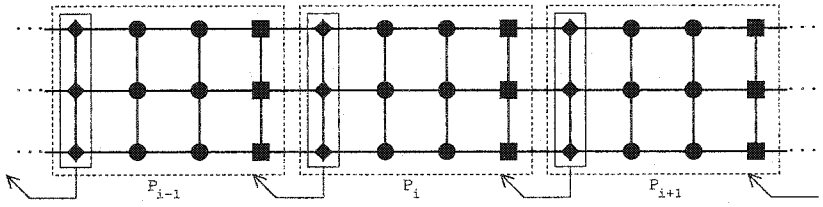
Entre chacune des itérations, une phase de communication est requise. D'abord, plusieurs informations locales doivent être échangées afin de faire les tests de convergence. Plusieurs statistiques comme le nombre de sommets déplacés, le déplacement minimal, maximal et moyen, en euclidien et dans la métrique sont échangées entre les processeurs afin de trouver les statistiques de déplacement du maillage global et de faire les tests de convergence. MPI fournit des fonctions de réduction permettant, par exemple, de calculer une valeur moyenne ou maximale sur tous les processeurs comme `MPI_Reduce()` ou `MPI_Allreduce()`. Toutefois, dans le cas présent, parmi les opérations à faire, on recherche des maximums, des minimums et des moyennes. Les opérations de réduction de MPI se prêtent mal aux cas nécessitant plusieurs opérations de réduction différente lors du même appel. Il était donc plus efficace de procéder avec la fonction `MPI_Allgather()` qui permet aux processeurs de connaître les extremums locaux et les moyennes locales de chacun des processeurs pour ensuite calculer les valeurs globales à tout le maillage. Ces valeurs sont alors utilisées pour l'affichage des statistiques et les tests de convergence.

Une fois cette phase de communication effectuée, une phase de synchronisation des frontières est requise. À la fin d'une itération impaire, chaque processeur envoie au processeur précédent les nouvelles coordonnées des sommets de sa frontière gauche. Cela

permettra de déplacer sécuritairement les sommets de la frontière droite lors de l'itération subséquente. À la fin d'une itération paire, les processeurs envoient les nouvelles coordonnées des sommets de leur frontière droite au processeur suivant. Les figures 3.9 et 3.10 montrent les sommets déplacés et les phases de communication des sommets frontières à chaque itération.



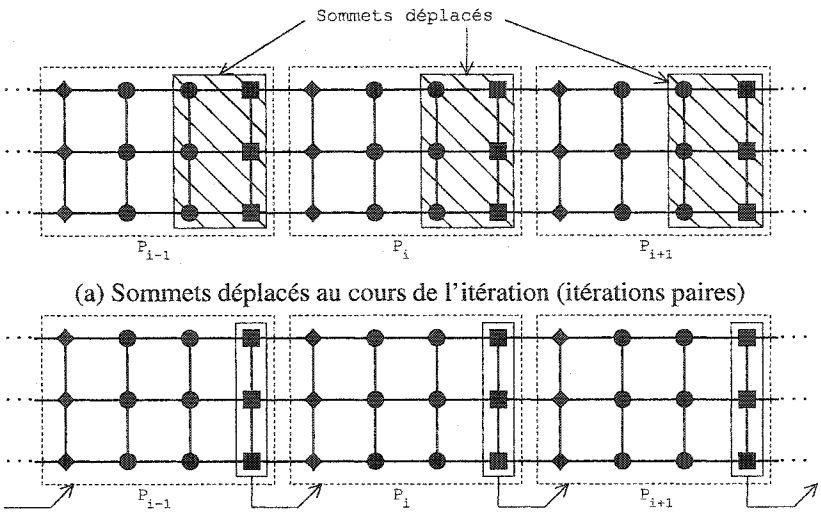
(a) Sommets déplacés au cours de l'itération (itérations impaires)



(b) Synchronisation de la frontière gauche (à la fin des itérations impaires)

FIGURE 3.9: Sommets déplacés et mis à jour lors des itération impaires

Cette opération de synchronisation des frontières nécessite une série de communications deux à deux. À la fin d'une itération impaire, tous les processeurs P_i envoient un message aux processeurs P_{i-1} et reçoivent un message du processeur P_{i+1}. À la fin d'une itération paire, les messages sont échangés dans le sens inverse. Il s'agit d'une communication directe deux à deux. Une simple combinaison `MPI_Send()` et `MPI_Recv()` fait donc l'affaire. Toutefois, comme on ne sait pas préalablement l'ordre dans lequel les processeurs feront les appels pour l'envoi et la réception des données, on ne voudrait pas créer de situation de verrous mortels. En outre, le volume de données à échanger peut être arbitrairement grand, selon la taille du maillage et des frontières. Il serait donc utile d'avoir un mode d'envoi et de réception non bloquant, mais synchronisé. Pour ce faire, la



(a) Sommets déplacés au cours de l'itération (itérations paires)

(b) Synchronisation de la frontière droite (à la fin des itérations paires)

FIGURE 3.10: Sommets déplacés et mis à jour lors des itération paires

combinaison `MPI_Issend()` et `MPI_Irecv()` est utilisée. Il faut toutefois alors s'assurer que les données ont bel et bien été reçues avant de les traiter et qu'elles ont bel et bien été envoyées avant d'effacer les tampons d'envoi. La fonction `MPI_Wait()` est utilisée afin d'attendre la complétion des requêtes de communication non bloquantes.

3.4.3 Synchronisation finale

À la toute fin du processus de déplacement des sommets, une étape de synchronisation finale est effectuée. Lors de cette étape, tous les processeurs autres que le processeur maître (de rang 0) doivent envoyer la position de tous leurs sommets au processeur maître. Ce dernier pourra alors mettre à jour le maillage global afin d'écrire le maillage adapté dans un fichier de sortie. La fonction de communication `MPI_Allgather()` est utilisée et permet à chaque processeur d'envoyer la position de ses sommets au processeur maître qui collecte ces informations.

3.5 Architecture du système

Afin d'implanter le nouvel algorithme de partitionnement et le nouvel algorithme de déplacement des sommets en parallèles, quelques changements importants ont été apportés à l'architecture de **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$. Premièrement, la classe `REMAILLEUR_3D_GENERIQUE` a été éliminée de la hiérarchie. Il existe maintenant une classe `REMAILLEUR_3D` qui définit l'interface et l'implantation du remailleur séquentiel. Le `REMAILLEUR_3D_PARALLELE` dérive du `REMAILLEUR_3D` et réimplante la fonction de déplacement des sommets afin d'inclure les opérations de partitionnement des sommets, de communication entre les itérations et finalisation du déplacement parallèle.

La figure 3.11 montre le diagramme de classes des remailleurs. Le `REMAILLEUR_3D` définit une liste des sommets et des itérateurs à ces sommets. Un itérateur permet d'itérer sur les sommets à déplacer durant les itérations paires et un autre durant les itérations impaires. Dans **OO**$\mathcal{RT}$, chacun des itérateurs pointe sur tous les sommets, mais permettent de les parcourir dans l'ordre inverse, une itération sur deux. Dans $\mathbb{P}$-**OO**$\mathcal{RT}$, après avoir partitionné les sommets, le `REMAILLEUR_3D_PARALLELE` possède quatre itérateurs. Deux itérateurs permettent d'itérer sur tous les sommets du maillage (dans deux ordres différents) et deux autres permettent d'itérer sur les sommets de la partition locale. Chacun de ces deux derniers pointeurs permettent respectivement d'accéder aux sommets à déplacer lors d'une itération paire et aux sommets à déplacer lors d'une itération impaire.

La figure 3.12 montre la séquence des appels lors du déplacement de sommets en parallèle. Ainsi, la fonction `deplacer_lesSommets()`, surchargée dans `REMAILLEUR_3D_PARALLELE`, appelle d'abord la fonction `partitionner_lesSommets()` qui crée le `PARTITIONNEUR` qui, à sont tour, construit les partitions. Les itérateurs du `REMAILLEUR_3D` sont alors redéfinis pour pointer sur les sommets de la partition locale du processeur et, par la suite, la fonction `deplacer_lesSommets()`

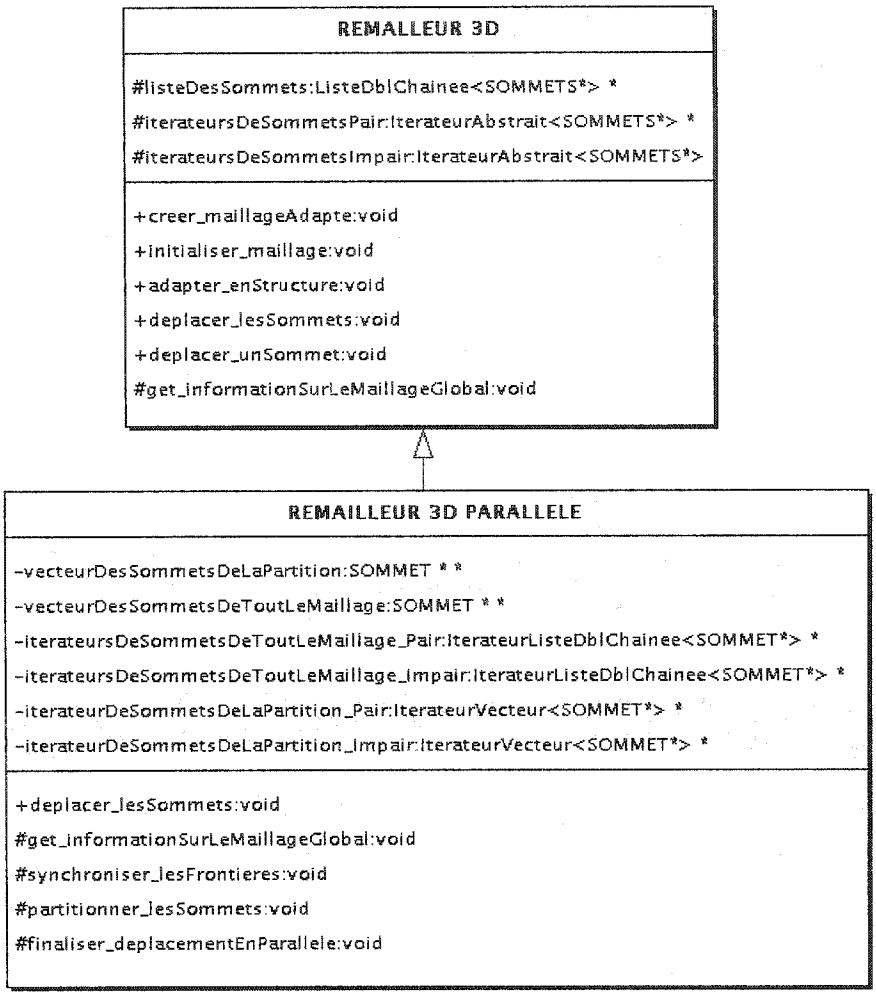


FIGURE 3.11: Diagramme de classes du REMAILLEUR_3D et du REMAILLEUR_3D_-PARALLELE.

du REMAILLEUR_3D est appelée pour faire le déplacement des sommets locaux. À la fin de chaque itération, les fonctions get_informationSurLeMaillageGlobal() et synchroniser_lesFrontieres() sont appelées pour communiquer et calculer les statistiques globales de déplacement et pour mettre à jour les sommets frontières qui ont été déplacés. Finalement, une fois le processus de déplacement terminé, la fonction finaliser_deplacementEnParallele() est appelée afin que le processeur maître mette à jour son maillage complet.

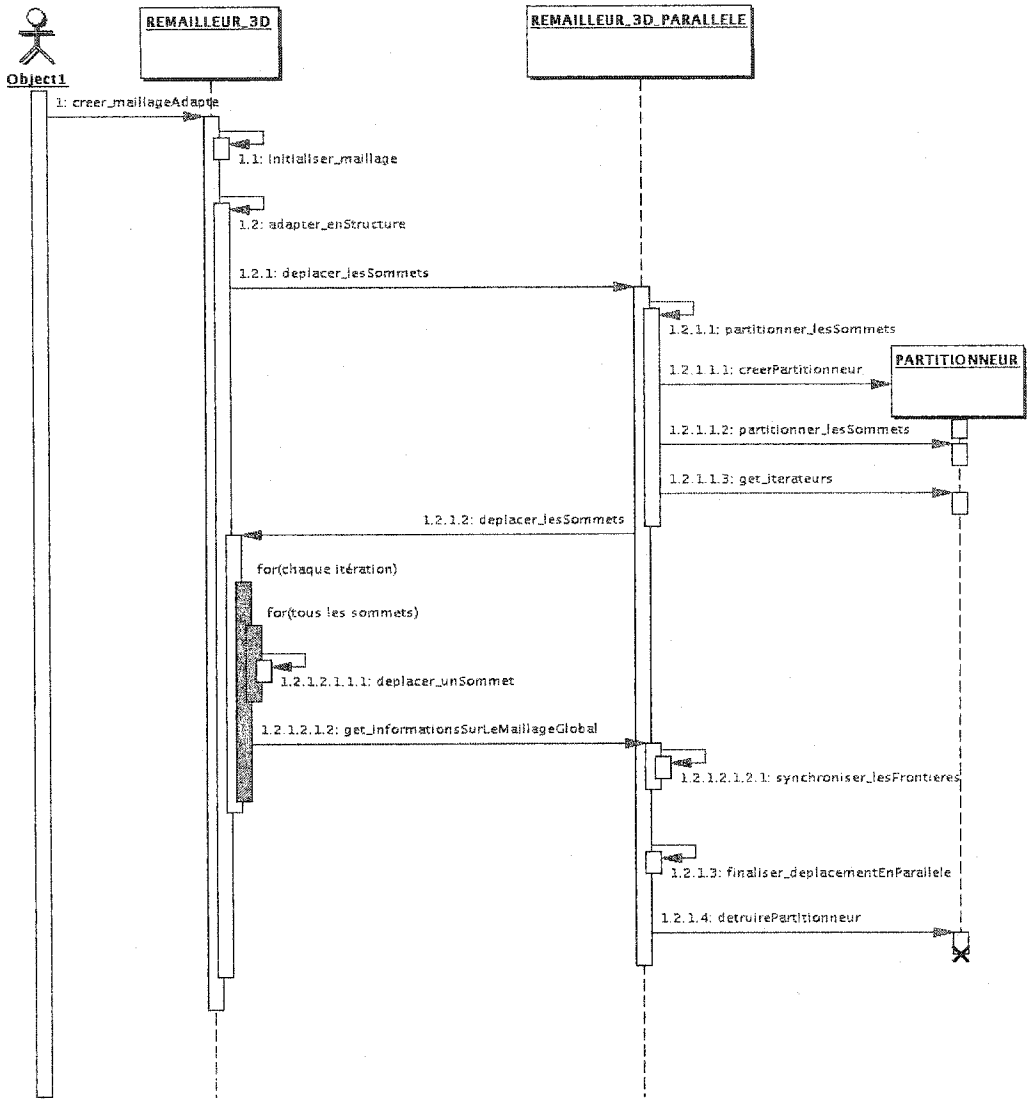


FIGURE 3.12: Diagramme de séquence du déplacement de sommets en parallèle.

3.6 Algorithme de repartitionnement dynamique

Le nouvel algorithme de partitionnement permet d'éliminer le problème d'éléments non conformes. Maintenant, dans le but d'améliorer l'équilibre de la charge de travail parmi les processeurs, l'algorithme de partitionnement sera légèrement modifié afin d'y ajouter

la possibilité de faire du repartitionnement dynamique.

3.6.1 Construction des blocs de sommets

L'algorithme de partitionnement utilisé dans le cadre du repartitionnement dynamique est pratiquement le même que celui présenté précédemment pour la partitionnement statique. La seule différence notable se situe au niveau de la construction des blocs. Dans l'algorithme présenté précédemment, chaque processeur peut avoir plusieurs blocs, alors que dans la cas présent, chaque processeur aura un seul bloc. Chacun des blocs sera donc de plus grande taille. Tout le reste de l'algorithme de partitionnement reste identique.

3.6.2 Architecture des `PARTITIONNEURs`

Afin d'introduire la fonctionnalité de repartitionnement dynamique, une hiérarchie de classes `PARTITIONNEUR` a été développée. Cette structure de classes est présentée à la figure 3.13. La classe `PARTITIONNEUR_GENERIQUE`, définit l'interface de base des `PARTITIONNEUR` et implante les fonctions communes. L'interface publique est composée de deux fonctions principales, `partitionner_lesSommets()` et `repartitionner_lesSommets()` ainsi que d'une série de fonctions permettant au client d'aller chercher les itérateurs créés. La fonction `partitionner_lesSommets()` est non virtuelle et implante le processus de partitionnement. Pour ce faire, elle appelle plusieurs autres fonctions, dont `construire_lesBlocs()` qui est virtuelle pure, car son comportement varie selon qu'il s'agisse ou non de repartitionnement dynamique.

La deuxième fonction principale est la fonction `repartitionner_lesSommets()`. Cette fonction est virtuelle pure. Dans la classe `PARTITIONNEUR_STATIQUE`, cette fonction retourne tout simplement la valeur « false », indiquant qu'aucun partitionne-

ment n'a été fait. Dans le cas où l'usager désire faire du repartitionnement dynamique, c'est la classe `PARTITIONNEUR_DYNAMIQUE` qui est instanciée et la fonction `repartitionner_lesSommets()` de cette classe se chargera de coordonner le repartitionnement.

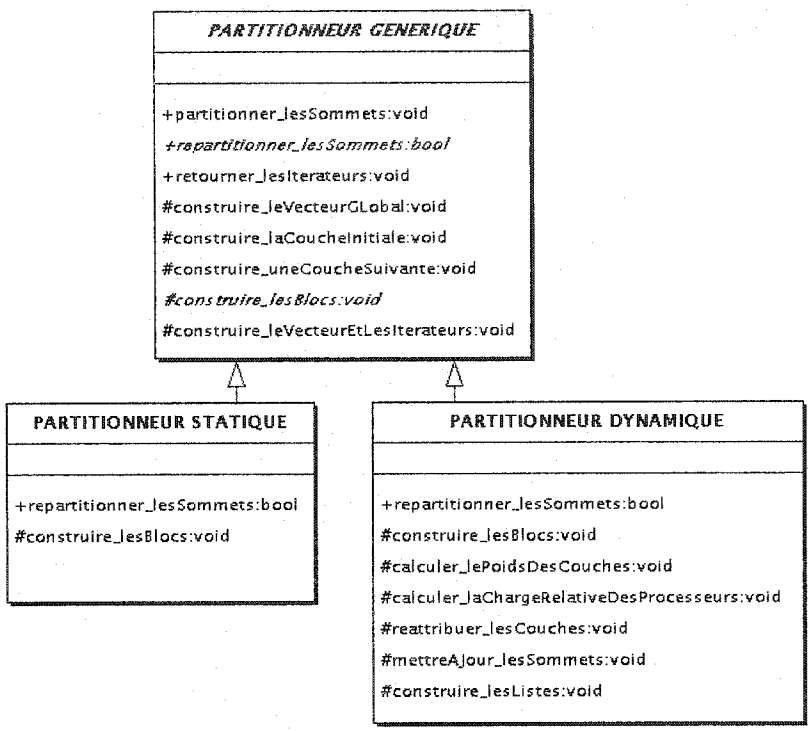


FIGURE 3.13: Diagramme des classes associées au partitionnement.

3.6.3 Développement d'une fonction de poids

Avant d'aller plus à fond dans l'algorithme de repartitionnement, il est essentiel de développer une fonction de poids associée à chaque sommet qui reflète la quantité de travail requise par ce sommet. Si la fonction de poids permet de prédire efficacement le travail futur à faire sur un sommet, il suffira alors de distribuer également le poids sur les processeurs afin d'avoir un bon équilibre de la charge. Dans **OO***RT*, la quantité de travail

associée à un sommet correspond au travail requis pour le déplacer. Or, le temps nécessaire pour déplacer un sommet est à peu près identique pour tous les sommets et n'est pas influencé par la magnitude du déplacement (c'est la même formule de déplacement qui est appliquée à tous les sommets). Cependant, tous les sommets ne sont pas déplacés systématiquement à chaque itération. Le processus de raffinement du critère d'arrêt local permet de ne déplacer que les sommets ayant le plus grand déplacement au début du processus itératif pour ensuite graduellement raffiner le critère local. Le poids d'un sommet devrait donc, en quelques sortes, refléter la probabilité qu'un sommet doive être déplacé au cours d'une itération.

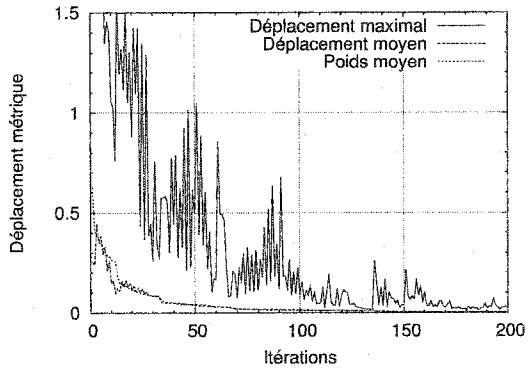
Normalement, plus un sommet se déplace au cours d'une itération, plus ses chances de se déplacer dans l'itération suivante sont fortes. Une approche permettant au sommet de conserver un historique de déplacement a donc été utilisée. Il s'agit en fait d'une méthode de sous-relaxation de la forme :

$$P_s^k = \alpha P_s^{k-1} + (1-\alpha) D_s^k$$

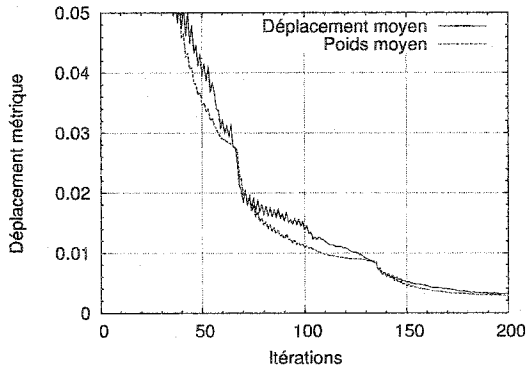
où P_s^k et D_s^k correspondent respectivement au poids et au déplacement du sommet s à l'itération k et où α est la constante de sous-relaxation. La valeur $\alpha = 0.5$ a été utilisée. Pour ce qui est du déplacement, il s'agit du déplacement adimensionnel du sommet (dans l'espace métrique).

Quelques tests préliminaires de cette fonction de poids ont montré un comportement intéressant du critère d'arrêt lors du processus de déplacement des sommets. Ce critère d'arrêt était initialement basé sur le déplacement maximal des sommets. À chaque itération, la valeur de déplacement du sommet s'étant le plus déplacé était comparé au critère d'arrêt et le processus était considéré comme ayant convergé si ce déplacement maximal était inférieur au critère d'arrêt. Or, la valeur du déplacement maximal oscille beaucoup entre chaque itération. Les figures 3.14 à 3.18 comparent l'évolution de trois différentes

valeurs candidates en tant que critère d'arrêt pour cinq différents cas tests. Pour chaque cas test, on présente deux graphiques (deux échelles différentes).

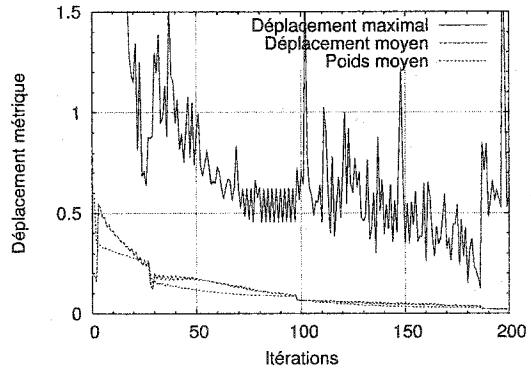


(a) Comparaison des trois critères

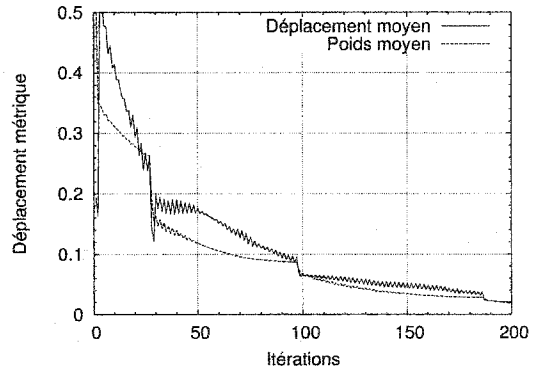


(b) Comparaison ente le déplacement moyen et la poids moyen

FIGURE 3.14: Comparaison des critères d'arrêt pour le cas test Arctg à 5625 sommets.



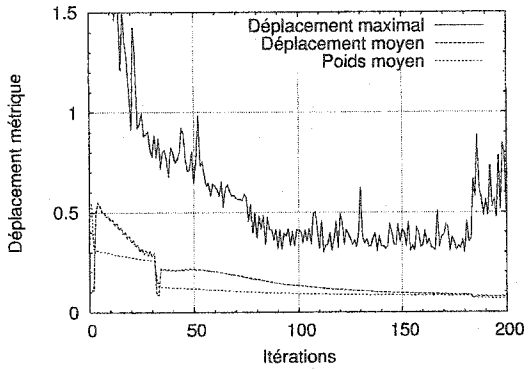
(a) Comparaison des trois critères



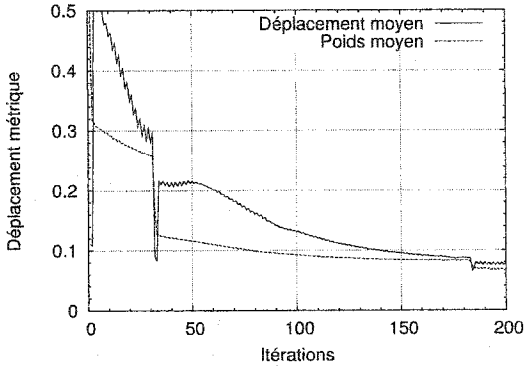
(b) Comparaison ente le déplacement moyen et la poids moyen

FIGURE 3.15: Comparaison des critères d'arrêt pour le cas test Arctg à 45000 sommets.

Le premier candidat correspond au déplacement maximal des sommets. Le second correspond au déplacement moyen des sommets. Finalement, le troisième correspond au poids moyen des sommets. On remarque immédiatement que le déplacement maximal est une donnée qui oscille énormément. Cette donnée semble donc plus ou moins fiable en tant que critère d'arrêt. Le déplacement moyen quant à lui, semble décroître de façon plus régulière. Le poids moyen des sommets qui, dans chacun des cinq cas tests, est la

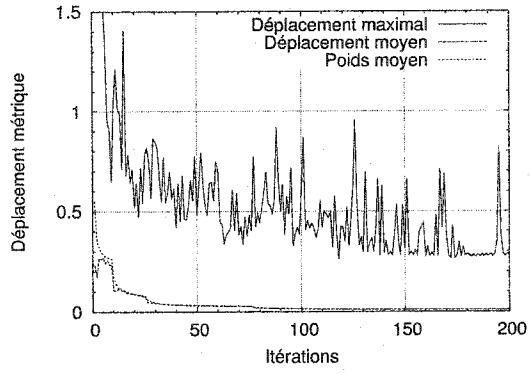


(a) Comparaison des trois critères

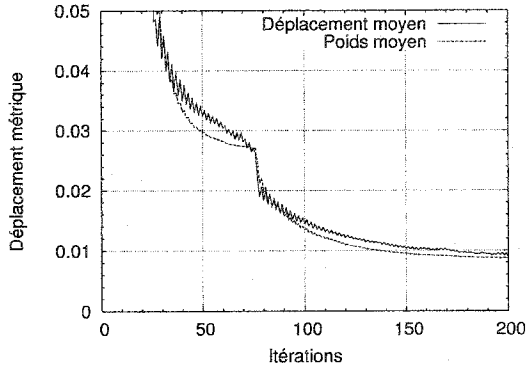


(b) Comparaison ente le déplacement moyen et la poids moyen

FIGURE 3.16: Comparaison des critères d'arrêt pour le cas test Arctg à 360000 sommets.



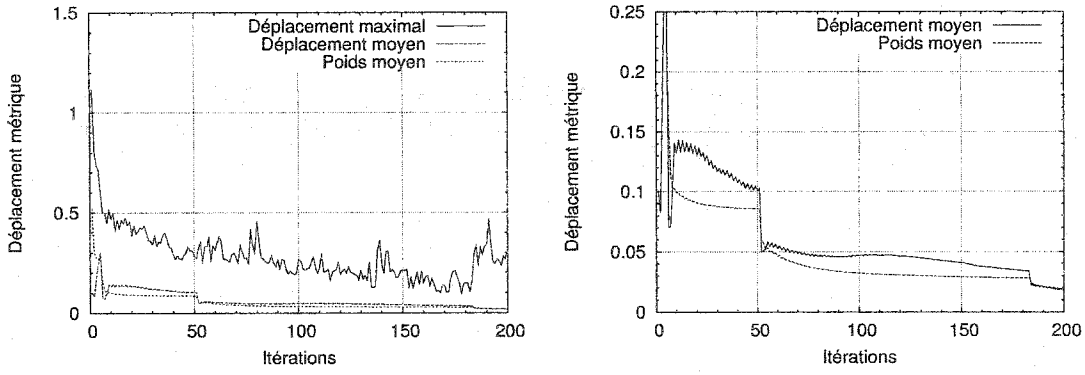
(a) Comparaison des trois critères



(b) Comparaison ente le déplacement moyen et la poids moyen

FIGURE 3.17: Comparaison des critères d'arrêt pour le cas test Ercoftac.

valeur qui décroît avec le plus de régularité, se comporte de façon quasi monotone. À la lumière de ces résultats, la fonction de poids, initialement définie pour $\mathbb{P}$-**OO**$\mathcal{RT}$ a été incorporée à **OO**$\mathcal{RT}$ en tant que critère d'arrêt au processus de déplacement des sommets. Dorénavant, ce critère est comparé au poids moyen des sommets plutôt qu'au déplacement maximal.



(a) Comparaison des trois critères

(b) Comparaison ente le déplacement moyen et la poids moyen

FIGURE 3.18: Comparaison des critères d'arrêt pour le cas test Dt151_Sweden.

3.6.4 Séquence de repartitionnement

La fonction `repartitionner_lesSommets()` se charge de faire le repartitionnement dynamique. Elle est appelée par le `REMAILLEUR_3D_PARALLELE` à intervalle régulier si l'usager a demandé de faire du repartitionnement dynamique. L'usager spécifie à quelle itération le premier repartitionnement doit avoir lieu et selon quel intervalle il doit avoir lieu par la suite. Selon que l'usager a demandé ou non de faire du repartitionnement dynamique, c'est le `PARTITIONNEUR_DYNAMIQUE` ou le `PARTITIONNEUR_STATIQUE` qui est instancié. La fonction `repartitionner_lesSommets()` retourne une valeur booléenne indiquant si du repartitionnement a effectivement été fait (« true ») ou non (« false »). En fonction de la valeur de retour, le `REMAILLEUR_3D_PARALLELE` sait s'il doit ou non mettre ses itérateurs à jour.

L'algorithme de la fonction `repartitionner_lesSommets()` est présenté à la figure 3.19. Cette fonction appelle plusieurs autres fonctions qui seront présentées par la suite.

La première étape du repartitionnement consiste à calculer le poids de chacune des couches. Pour ce faire, la fonction `calculer_poidsDesCouches()` procède en

```
Appeler calculer_poidsDesCouches();
Appeler calculer_chargeRelative(Répartition originale);
Calculer le déséquilibre de la charge;
si Le déséquilibre est inférieur au seuil alors
    On ne repartitionne pas (retourner « false »);
fin
Appeler reattribuer_LesCouches();
si On n'a pas trouvé de meilleure répartition de la charge alors
    On ne repartitionne pas (retourner « false »);
fin
Appeler calculer_chargeRelative(Nouvelle répartition);
Calculer le déséquilibre de la charge obtenu avec la nouvelle répartition;
si Le déséquilibre de charge n'est pas amélioré avec la nouvelle répartition
alors
    On ne repartitionne pas (retourner « false »);
fin
Appeler migrer_lesSommetsDeplaces();
Appeler construire_lesIterateurs();
Signifier que le partitionnement a été fait (retourner « true »)
```

FIGURE 3.19: Algorithme de la fonction `repartitionner_lesSommets()`.

deux sous étapes. D'abord, chaque processeur calcule le poids total de chacune de ses couches locales. Ensuite, une ronde de communication est effectuée. Durant cette communication, les processeurs s'échangent le poids de leurs couches locales de façon à ce que chaque processeur connaisse le poids de toutes les couches du maillage. Il s'agit d'une communication de type « tous à tous » où chaque processeur envoie aux autres le poids de ses couches. Comme le nombre de couches par processeur varie, la fonction `MPI_Allgatherv()` est utilisée pour l'échange des messages.

La deuxième étape du repartitionnement consiste à calculer la charge relative de chacun des processeurs. La fonction `calculer_chargeRelative()` effectue ce travail. Chaque processeur calcule la charge relative de tous les processeurs à partir du poids des couches obtenu à l'étape précédente. À l'aide d'un index indiquant à quel processeur appartient chacune des couches, le poids total des sommets de chaque processeur est

calculé (en faisant la somme de ses couches) ainsi que le poids total de tous les processeurs. Puis, la charge relative de chaque processeur P_i est calculée. Il s'agit du rapport entre la somme des poids des sommets de ce processeur sur la somme des poids de tous les sommets. À partir de la charge relative de chaque processeur, `repartitionner_lesSommets()` calcule alors le déséquilibre de la charge. Le déséquilibre de la charge se calcule de la façon suivante : si C_{max} est la charge relative du processeur le plus occupé et $C_{\text{idéal}}$ est la charge relative idéale correspondant au cas où tous les processeurs ont la même charge relative, le déséquilibre de charge D est obtenu en calculant le rapport suivant :

$$D = \frac{C_{\text{max}} - C_{\text{idéal}}}{C_{\text{idéal}}}$$

Si le déséquilibre de charge est inférieur à un certain seuil, alors il n'est pas utile de repartitionner, car la charge est déjà équilibrée. La fonction `repartitionner_lesSommets()` s'interrompt en signalant que le repartitionnement n'a pas été effectué.

La troisième étape du repartitionnement consiste à redistribuer les couches de manière à équilibrer la charge. Cette tâche est effectuée par la fonction `reattribuer_LesCouches()`. Le principe est très simple. Le but est de répartir la charge totale le plus également possible. On parcoure donc les couches de 0 à n. On sait qu'une partition doit toujours contenir au moins deux couches (une frontière gauche et une frontière droite). Ainsi, les deux premières couches sont ajoutées dans la partition courante et leur poids est ajouté au poids de la partition courante. Puis, on ajoute des couches à la partition courante tant que le poids de cette partition n'a pas dépassé le poids cible. Le poids cible correspond à un poids également réparti entre les processeurs. Lorsque le poids cible est dépassé, on vérifie si la dernière couche ajoutée à la partition doit être enlevée ou non. Pour ce faire, on vérifie dans quel cas la différence entre le poids cible et le poids de la partition courante (en valeur absolue) est inférieur. C'est le cas qui sera privilégié. La nouvelle association entre les couches et les processeurs est stockée dans une structure

temporaire par la fonction `repartitionner_lesSommets()`.

La quatrième étape consiste à calculer la charge relative des processeurs avec la nouvelle distribution des couches. Cela se fait avec la fonction `calculer_chargeRelative()`. Étant donné que les couches peuvent être grandes et donc la granularité du partitionnement être faible, il est tout à fait possible que le nouveau partitionnement soit de moindre qualité ou de qualité égale (c'est le même partitionnement) que le partitionnement initial. C'est pourquoi on recalcule la charge relative des processeurs et le déséquilibre de la charge. Si le nouveau déséquilibre de la charge est plus grand ou égal, le nouvel index associant les couches aux processeurs est éliminé, car on reprend alors le partitionnement original. La fonction `repartitionner_lesSommets()` se termine alors en signifiant que le repartitionnement n'a pas été fait.

Arrivé à la cinquième étape, on sait qu'un nouveau partitionnement a été calculé et qu'il est meilleur que le partitionnement initial. Il reste donc à effectuer le partitionnement comme tel. Il faut d'abord s'assurer que les sommets qui changent de propriétaire soient mis à jour sur leur nouveau propriétaire. Par exemple, si le sommet s appartient au processeur p_1 avant le repartitionnement et qu'il appartient maintenant au processeur p_3 après le repartitionnement, p_1 doit envoyer à p_3 la position à jour de s de façon à ce que le maillage reste cohérent. Ceci est fait dans la fonction `migrer_lesSommetsDeplaces()`. Chaque processeur identifie les sommets qu'il doit envoyer aux autres et ceux qu'il doit recevoir des autres. Pour ce faire, pour chaque couche de sommets, on détermine le processeur d'origine (processeur où la couche était située avant le repartitionnement) et le processeur de destination (processeur où la couche sera située après le repartitionnement). Chaque processeur détermine alors la liste des sommets à envoyer à chacun des autres et la liste de sommets à recevoir de chacun des autres. Par la suite, la communication peut être effectuée. Il s'agit d'un schéma de communication de type « tous à tous personnalisées » réalisé à l'aide de la fonction `MPI_Alltoallv()`.

La sixième et dernière étape consiste à reconstruire le vecteur des sommets de la partition ainsi que les différents itérateurs. La fonction `construire_lesIterateurs()` utilisée est la même qu'au moment du partitionnement initial. Une fois le travail terminé, `repartitionner_lesSommets()` signifie au `REMAILLEUR_3D_PARALLELE` que le partitionnement a bel et bien été effectué. `REMAILLEUR_3D_PARALLELE` est alors responsable de mettre à jour ses itérateurs en interrogeant le `PARTITIONNEUR_DYNAMIQUE` et le processus de déplacement des sommets peut reprendre.

3.7 Utilisation de la mémoire

Ainsi, les nouveaux algorithmes de partitionnement et de repartitionnement dynamique devraient permettre à la fois d'éliminer le problème d'éléments non conformes et améliorer la performance de l'application parallèle. Pour ce qui est de la mémoire cependant, c'est moins clair. Il serait possible, sans trop de difficulté, de faire en sorte que les différents processeurs libèrent la mémoire allouée pour les sommets qui ne sont pas sur la partition courante ou qui ne sont pas sur les couches voisines de la partition courante. Cependant, il n'est pas évident que cela permettrait d'améliorer l'utilisation de la mémoire pour, par exemple, traiter des maillages de plus grande taille, car, de toutes façons, les algorithmes de partitionnement sont conçus pour tirer avantage du fait que tous les processeurs connaissent tous les sommets. Par exemple, durant la division des sommets en couches, aucune communication n'est requise entre les processeurs, ce qui ne serait pas le cas si les sommets étaient distribués sur tous les processeurs. Toute la partie concernant la construction des couches serait grandement complexifiée et moins performante si le maillage était distribué. Il en va de même pour la construction des blocs et leur assignation à une partition. Une fois les couches et les partitions construites toutefois, il serait possible de désallouer la mémoire allouée aux sommets qui ne font partie de la partition, mais cela ne change rien au fait qu'au début du partitionnement, tous les

sommets sont connus de tous les processeurs.

De plus, tout l'algorithme de repartitionnement dynamique serait beaucoup plus complexe si le maillage était distribué. Lorsque des sommets changent de partition, on ne pourrait plus se contenter de mettre à jour leur position sur leur nouvelle partition. Il faudrait alors communiquer toute l'information nécessaire pour reconstruire les zones de maillage affectées ce qui inclut la position des sommets, les informations de connectivité (arêtes, éléments), le support topologique des sommets, etc. Tout ceci se traduirait par, non seulement une complexité accrue du code, mais par une perte de performance due à une augmentation des communications requises pour la reconstruction des partitions.

En outre, $\mathbb{P}$-**OO**$\mathcal{RT}$ se base sur la librairie **OO**$\mathcal{RT}$ elle-même basée sur la libraire **PIRATE**[1][2] (voir Labbé *et al.* (2000)). Tel que vu dans l'introduction, **PIRATE** est une librairie écrite en C++ définissant des structure de données et des méthodes pour gérer des modèles géométriques et topologiques, des maillages et des solutions associées au maillage. Elle définit, entre autres, un analyseur de syntaxe qui permet de lire et d'écrire les fichiers d'entrée et de sortie des applications basées sur **PIRATE**. Ainsi, lors de la phase de lecture et d'écriture des fichiers, il est nécessaire qu'un des processeurs connaisse l'entièreté du maillage de façon à gérer les entrées et les sorties. Si l'utilisation de la mémoire devait être optimisée, malgré tous les efforts déployés, il reste nécessaire qu'un processeur connaisse tout le maillage, ce qui limite les possibilités d'économie de la mémoire. Ce problème pourrait possiblement être contourné en utilisant les fonctions d'entrée et de sortie de MPI-I/O, mais cela impliquerait une réécriture du module faisant la lecture et l'écriture de fichiers.

Finalement, les différents cas tests utilisés pour tester l'application parallèle ne justi-

[1] www.polymtl.ca/grmiao/grmiao/Pir/doc/manuel/usr/usr/
[2] www.polymtl.ca/grmiao/grmiao/Pir/doc/manuel/fic/fic/

fient pas une optimisation de l'utilisation de la mémoire au détriment de la performance puisque l'espace mémoire utilisé reste, malgré tout, suffisamment petit et ne cause pas de problème sur les architectures utilisées. Ces différentes raisons expliquent pourquoi l'optimisation de l'utilisation de la mémoire a été négligée lors de la conception de ces nouveaux algorithmes.

CHAPITRE 4

ANALYSE DES RÉSULTATS

4.1 Introduction

Le chapitre précédent a permis de présenter et de décrire en détail les changements apportés dans $\mathbb{P}$-**OO**$\mathcal{RT}$ afin de régler les problèmes identifiés au chapitre 2. Notamment, un nouvel algorithme de partitionnement des sommets et un algorithme de repartitionnement dynamique ont été conçus afin de régler les problèmes d'éléments non conformes et de performance parallèle décevante. Ce chapitre présente les tests effectués, les résultats obtenus ainsi qu'une analyse de ces résultats afin de voir si les problèmes identifiés préalablement ont été réglés.

4.2 Environnement des tests

Avant de présenter les résultats et de les analyser, il est nécessaire de présenter l'environnement dans lequel les tests ont été effectués. Cette section présente donc les architectures sur lesquelles les tests ont été effectués, les cas tests utilisés, les critères de comparaison entre la version séquentielle et la version parallèle ainsi que les différentes mesures de performance effectuées.

4.2.1 Architectures testées

En premier lieu, les architectures utilisées sont sensiblement les mêmes que pour le prototype. Il s'agit de HEYSE et de CHARYBDE qui ont déjà été présentées au chapitre 2. Toutefois, depuis l'étape des tests sur le prototype, quelques changements leur ont été apportés. Le changement le plus notable est le passage de 16 à 8 noeuds pour HEYSE. En outre, certaines mises à jour logicielles ont été apportées. Tous les détails des architectures utilisées peuvent être trouvés dans les tableaux 4.1 et 4.2.

TABLEAU 4.1 Configuration matérielle des architectures de tests.

Nom	Nbre de noeuds	Proc. par noeud	Type de proc.	Mém. par noeuds	Config. réseau
HEYSE	8	1	AMD Athlon, 1.4 GHz	1 Go	Ethernet
CHARYBDE	8	4	Pentium III Xeon, 700 MHz	4 Go	Myrinet[1]

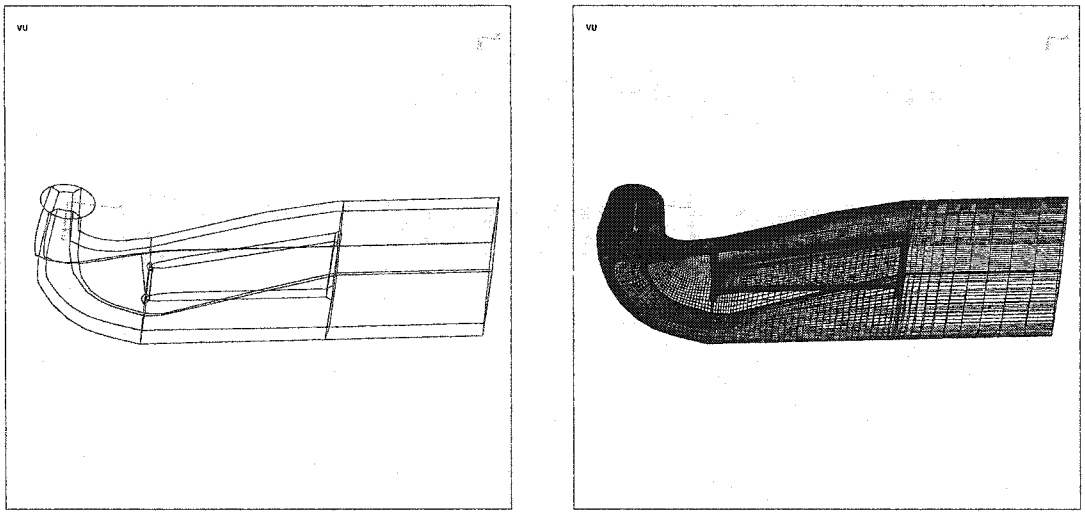
TABLEAU 4.2 Configuration logicielle des architectures de tests.

Nom	Système d'exploitation	Compilateur C++	Implantation MPI
HEYSE	Linux 2.4.18	gcc 2.95.3	LAM 6.5.9
CHARYBDE	Linux 2.4.20	gcc 3.2.1	MPICH-GM 1.2.5..9

4.2.2 Présentation des cas tests

Les cas tests M1 à M4, présentés au chapitre 2, ont été repris et un cinquième, M5, a été ajouté. Il s'agit d'un aspirateur à une pile maillé avec 8 blocs structurés nommé Dt151_-Sweden. La figure 4.1 montre la géométrie et le maillage généré pour ce cas test. Les autres cas tests ont déjà été présentés à la figure 2.6 au chapitre 2. Le tableau 4.3 présente les informations pertinentes de chacun des cinq cas test qui sont utilisés pour tester la nouvelle version de IP-OORT.

[1] http ://www.myri.com/



(a) Géométrie (b) Maillage

FIGURE 4.1: Cas test Dt151_Sweden.

TABLEAU 4.3 Présentation des cas tests pour les nouveaux algorithmes.

Numéro du test	Nom du cas test	Nombre de sommets	Nombre d'arêtes	Nombre d'éléments
M1	Arctg	5625	15900	4704
M2	Arctg	45000	131100	41209
M3	Arctg	360000	1064400	344619
M4	Ercoftac	127982	377829	122004
M5	Dt151_Sweden	215828	637810	206312

4.2.3 Critère de comparaison entre OO$\mathcal{RT}$ et IP-OO$\mathcal{RT}$

Lors des tests effectués sur le prototype de IP-OO$\mathcal{RT}$, le critère de comparaison pour établir la performance de IP-OO$\mathcal{RT}$ par rapport à OO$\mathcal{RT}$ consistait simplement à exécuter le même nombre d'itérations dans les deux cas et de comparer les temps d'exécution. Cela était justifié par le fait que le nombre de sommets déplacés à chaque itération était le même et la somme de travail faite par OO$\mathcal{RT}$ et IP-OO$\mathcal{RT}$ pouvait être considérée équivalente. Toutefois, dans la nouvelle version de IP-OO$\mathcal{RT}$, le nombre de sommets déplacés à chaque itération n'est pas nécessairement le même (itérations paires et impaires) et il est beaucoup plus difficile de s'assurer que les deux applications

effectuent la même somme de travail ou, du moins, une somme comparable.

L'approche retenue est donc de mesurer le temps requis pour atteindre le même niveau de convergence. Il suffit donc de mettre un nombre maximal d'itérations suffisant pour que ce soit le critère d'arrêt qui provoque l'arrêt du processus de déplacement des sommets et on utilise le même critère d'arrêt pour **OO**$\mathcal{RT}$ et pour $\mathbb{P}$-**OO**$\mathcal{RT}$.

Cette comparaison constitue en outre une technique plus juste, car elle pourra tenir compte du fait que $\mathbb{P}$-**OO**$\mathcal{RT}$ améliore ou détériore la vitesse de convergence. En effet, rien ne garantit que pour la même somme de travail, i.e. le même nombre de déplacements de sommet, le niveau de convergence sera le même puisque la solution au cours d'une itération dépend de l'ordre de parcours des sommets. Ainsi, le nouveau critère de comparaison permettra de tenir compte de l'influence de $\mathbb{P}$-**OO**$\mathcal{RT}$ sur la vitesse de convergence.

On peut alors se demander pourquoi un tel critère de convergence n'a pas été utilisé lors des tests sur le prototype de $\mathbb{P}$-**OO**$\mathcal{RT}$. La raison est très simple. Il faut se rappeler qu'au cours du développement des nouveaux algorithmes de $\mathbb{P}$-**OO**$\mathcal{RT}$, un nouveau critère d'arrêt a été développé. L'ancien critère, basé sur le déplacement maximal des sommets avait une variabilité très grande et il était difficile de s'y fier (voir les figures 3.14 à 3.18).

Un deuxième critère de comparaison entre **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$ sera la qualité des maillages obtenus. Pour ce faire, il suffira de comparer les longueurs minimales, maximales et moyennes ainsi que l'écart type des arêtes dans l'espace de la métrique entre la version séquentielle et la version parallèle pour voir si la qualité des maillages est équivalente.

4.2.4 Mesures de performance

Pour mesurer les performances, de nombreuses mesures de temps seront effectuées. La fonction `MPI_Wtime()` est utilisée à cette fin. Chaque processeur maintient un total appelé « temps de calcul » qui correspond au temps passé à faire du calcul. Il s'agit du temps passé au cours de chaque itération entre l'étape de synchronisation des frontières de l'itération précédente et l'étape de communication des statistiques globales de l'itération courante. Il s'agit donc du temps passé à faire uniquement du calcul.

Plusieurs autres mesures de temps sont effectuées : le temps de partitionnement initial, le temps de repartitionnement, le temps de communication des statistiques de convergence, le temps de synchronisation des frontières, le temps de repartitionnement et le temps de finalisation de l'adaptation parallèle. De plus, à chaque itération, les processeurs doivent se synchroniser au niveau de la communication des statistiques (synchronisation par une communication collective `MPI_Allgather()`). Ainsi, le temps de communication des statistiques devrait être plus long sur les processeurs moins occupés, car ils doivent attendre que les autres processeurs soient prêts à échanger l'information. On calcule donc le temps d'attente en faisant la différence entre le temps de communication le plus long et le temps de communication le plus court, ce qui donne une idée de la surcharge du processeur le plus occupé par rapport au processeur le moins occupé.

4.3 Qualité des maillages obtenus

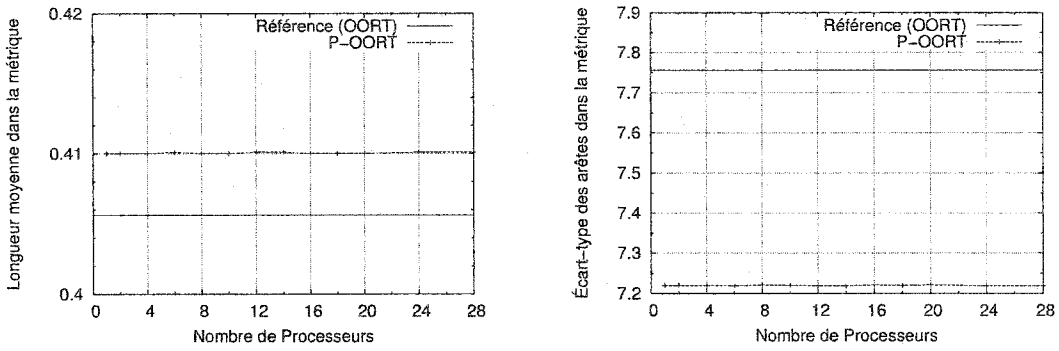
Dans le prototype de IP-OO$\mathcal{RT}$ présenté au chapitre 2, le principal problème lié à la qualité du maillage était la création d'éléments non conformes. Les nouveaux algorithmes devaient permettre d'éliminer ce problème. Pour le vérifier, à la fin du processus d'adaptation, une portion du code de IP-OO$\mathcal{RT}$ consiste à parcourir la liste des éléments du

maillage et à vérifier la validité de chacun. Sur ce point, tant le nouvel algorithme de partitionnement statique que l'algorithme de repartitionnement dynamique fonctionne. Aucun élément non conforme n'a été créé et ce, pour tous les cas tests et peu importe le nombre de processeurs utilisés.

Maintenant, qu'en est-il de la qualité du maillage dans $\mathbb{P}$-**OO**$\mathcal{RT}$ par rapport à **OO**$\mathcal{RT}$? La mesure de qualité est liée à la longueur des arêtes dans la métrique. En effet, le but du processus d'adaptation est d'obtenir des arêtes qui sont toutes de longueur unitaire dans l'espace métrique. Toutefois, il important de noter que dans un maillage structuré, ce but ne peut pas toujours être atteint puisque les opérations de raffinement et de déraffinement d'arêtes ne sont pas permises. Seul le déplacement de sommets est permis et si le maillage n'est pas assez dense (ou trop dense) le déplacement de sommets, seul, ne pourra pas permettre d'atteindre ce but. Ainsi, pour vérifier si $\mathbb{P}$-**OO**$\mathcal{RT}$ produit des maillages de bonne qualité il faut vérifier si la longueur métrique des arêtes s'est approchée ou éloignée du but visé par rapport à ce qui a été obtenu dans **OO**$\mathcal{RT}$, mais il est fort possible que la longueur obtenue par **OO**$\mathcal{RT}$ (et donc par $\mathbb{P}$-**OO**$\mathcal{RT}$) soit loin de la valeur visée si le maillage est trop ou (pas assez) dense, ce qui ne peut être réglé uniquement par le déplacement de sommets.

En résumé, pour vérifier si $\mathbb{P}$-**OO**$\mathcal{RT}$ affecte négativement la qualité des maillages obtenus par rapport à **OO**$\mathcal{RT}$, il faut vérifier que la longueur métrique moyenne des arêtes ne s'est pas éloignée du but visé (valeur unitaire) et que l'écart type n'a pas augmenté. Les figures 4.2 à 4.5 illustrent cette comparaison pour les cas tests ercoftac (M4) et Dt151_Sweden (M5). Chaque figure correspond à un cas test et un algorithme particulier. Ainsi, les deux premières (4.2 et 4.3) correspondent aux résultats obtenus pour chacun des deux cas tests dans le cas où le repartitionnement dynamique n'est pas utilisé, i.e., le cas où on ne fait que du partitionnement statique. Les deux suivantes (4.4 et 4.5) correspondent toujours aux mêmes deux cas tests, mais, cette fois-ci, le reparti-

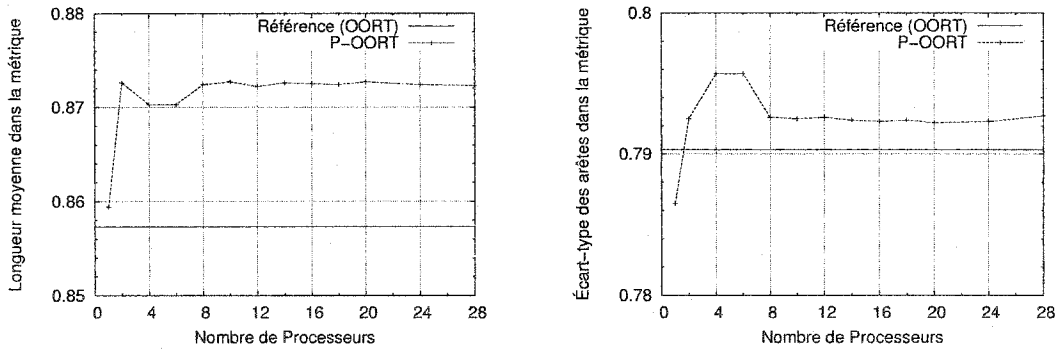
tionnement dynamique est utilisé. Dans chaque figure, on retrouve deux graphiques. Le premier donne la longueur moyenne des arêtes dans l'espace métrique et le deuxième donne l'écart type sur cette longueur. Sur chaque graphique, un trait plein illustre la valeur obtenue avec **OO**$\mathcal{RT}$ pour fins de comparaison.



(a) Longueur moyenne des arêtes dans la métrique

(b) Écart-type sur le longueur des arêtes dans la métrique

FIGURE 4.2: Influence de IP-**OO**$\mathcal{RT}$ sur la qualité du maillage obtenu pour le cas test Ercoftac (M4) et l'algorithme de partitionnement statique.

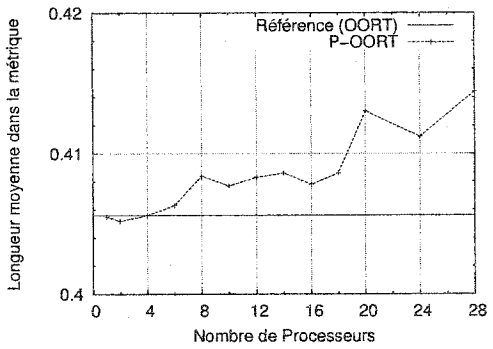


(a) Longueur moyenne des arêtes dans la métrique

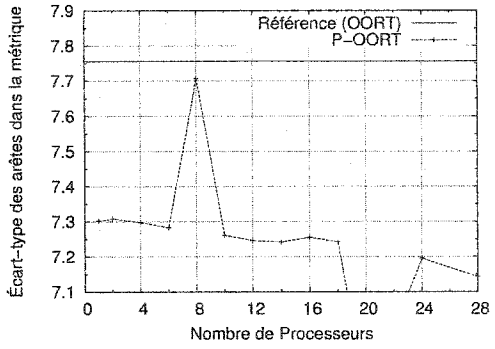
(b) Écart-type sur le longueur des arêtes dans la métrique

FIGURE 4.3: Influence de IP-**OO**$\mathcal{RT}$ sur la qualité du maillage obtenu pour le cas test Dt151_Sweden (M5) et l'algorithme de partitionnement statique.

En examinant les figures 4.2 et 4.3, on remarque que la qualité du maillage, dans la plupart des cas, est très légèrement meilleure dans IP-**OO**$\mathcal{RT}$. En effet, dans les deux cas, la longueur moyenne des arêtes est un peu plus près de la longueur unitaire (environ 2% plus près). De plus, dans le cas test ercoftac, l'écart type sur la longueur des arêtes

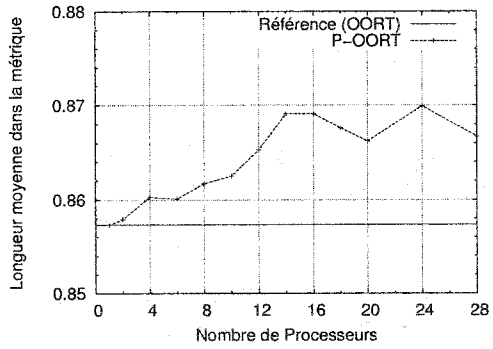


(a) Longueur moyenne des arêtes dans la métrique

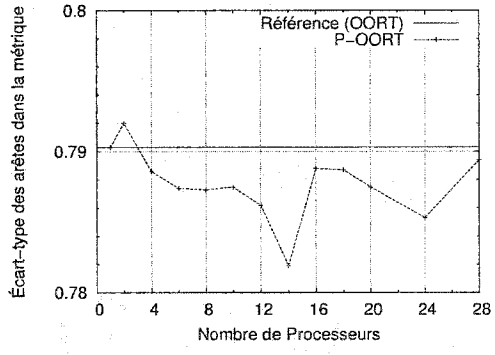


(b) Écart-type sur le longueur des arêtes dans la métrique

FIGURE 4.4: Influence de IP-**OO**$\mathcal{RT}$ sur la qualité du maillage obtenu pour le cas test Ercoftac (M4) et l'algorithme de repartitionnement dynamique.



(a) Longueur moyenne des arêtes dans la métrique



(b) Écart-type sur le longueur des arêtes dans la métrique

FIGURE 4.5: Influence de IP-**OO**$\mathcal{RT}$ sur la qualité du maillage obtenu pour le cas test Dt151_Sweden (M5) et l'algorithme de repartitionnement dynamique.

est également sensiblement meilleur (environ 7% plus bas). Seul l'écart type dans le test Dt151_Sweden semble moins bon, puisqu'il a augmenté d'environ 0.25%. Dans l'ensemble, il semble donc que IP-**OO**$\mathcal{RT}$, dans le cas du partitionnement statique, n'influence pas, ou alors très peu, la qualité des maillages par rapport à **OO**$\mathcal{RT}$. On peut toutefois remarquer que la longueur métrique moyenne des arêtes dans le cas test Ercoftac est tout de même assez loin du but visé (longueur unitaire). Ceci est simplement lié au fait que la densité du maillage ne satisfait pas la métrique, ce qui ne peut pas être corrigé uniquement par du déplacement de sommets. L'important dans ce cas-ci est de

voir la différence entre les résultats obtenus par OO$\mathcal{RT}$ et $\mathbb{P}$-OO$\mathcal{RT}$, différence somme toute négligeable.

Dans le cas des figures 4.4 et 4.5, il semble que les résultats soient très similaires. Dans tous les cas, la longueur moyenne des arêtes s'est légèrement améliorée par rapport à la valeur séquentielle étant donnée que la valeur parallèle est un peu plus prêt de la valeur unitaire que la valeur séquentielle. Les variations restent minimes toutefois, soit environ 2% en moyenne. L'algorithme de repartitionnement dynamique ne semble donc pas, lui non plus, affecter négativement la qualité du maillage obtenu en sortie.

Les mêmes tests ont également été faits sur les trois premiers cas tests, correspondant à la géométrie Arctg maillée à trois niveaux de granularité différents et, bien que la variabilité sur la longueur et sur l'écart type était légèrement supérieure ($\pm 5\%$), ces tests ont confirmé que $\mathbb{P}$-OO$\mathcal{RT}$ n'influençait pas de façon significative la qualité des maillages obtenus.

4.4 Performance

La présente section a pour but d'analyser les performances des nouveaux algorithmes de partitionnement et de les comparer aux performances obtenues avec le prototype. Dans un premier temps, une comparaison sera faite entre le prototype, le nouvel algorithme de partitionnement des sommets et le nouvel algorithme de repartitionnement dynamique. Pour ce qui est de l'algorithme de partitionnement statique, selon l'algorithme développé et présenté au chapitre 3, lors de la construction des blocs, l'usager peut spécifier la taille de chacun des blocs en indiquant le nombre de couches internes à mettre dans chaque bloc. Pour la première série de tests, ce nombre sera fixé à 0. L'influence de cette valeur sera analysée plus loin. Ensuite, lorsque l'usager choisit de faire du repartitionnement dynamique, il a la possibilité de choisir la fréquence à laquelle on repartitionne

le domaine. Dans la première série de tests, la fréquence choisie est de repartitionner à toutes les 10 itérations. L'influence de la fréquence de repartitionnement sera analysée plus loin.

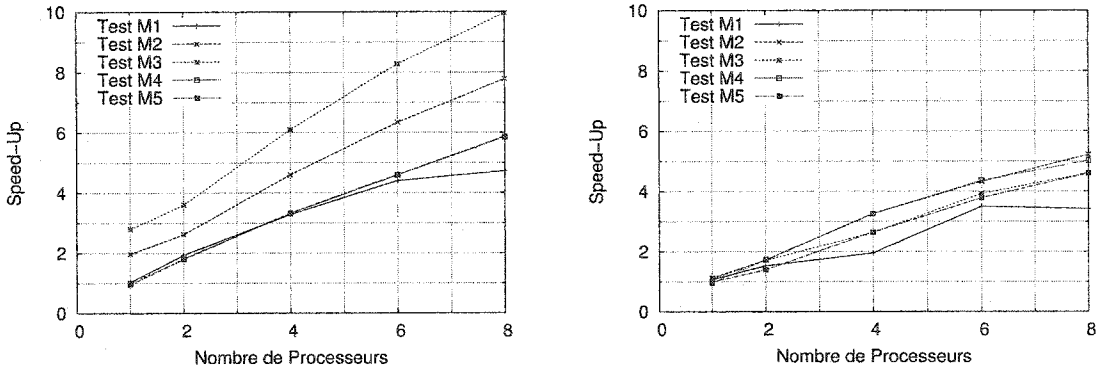
4.4.1 Comparaison avec le prototype

Ainsi, deux tests ont été effectués sur chacun des cas tests lors de la première série de tests. Dans le premier cas, l'algorithme de partitionnement statique a été utilisé et dans le deuxième cas, l'algorithme de repartitionnement dynamique a été utilisé. Le tableau 4.4 présente les temps d'exécution de **OO**$\mathcal{RT}$ sur chacun des cinq cas tests pour chacune des deux architectures testées. Ces temps serviront de référence pour les calculs de speed-up de $\mathbb{P}$-**OO**$\mathcal{RT}$. Les figures 4.6 et 4.7 présentent les speed-up obtenus pour **OO**$\mathcal{RT}$ pour chacune des deux architectures et pour chacun des cinq cas tests. Les figures 4.8 à 4.12 permettent, quant à elles, de comparer les nouveaux algorithmes avec le prototype en isolant les données de chaque cas test.

TABLEAU 4.4 Temps d'exécution de **OO**$\mathcal{RT}$ sur HEYSE et CHARYBDE.

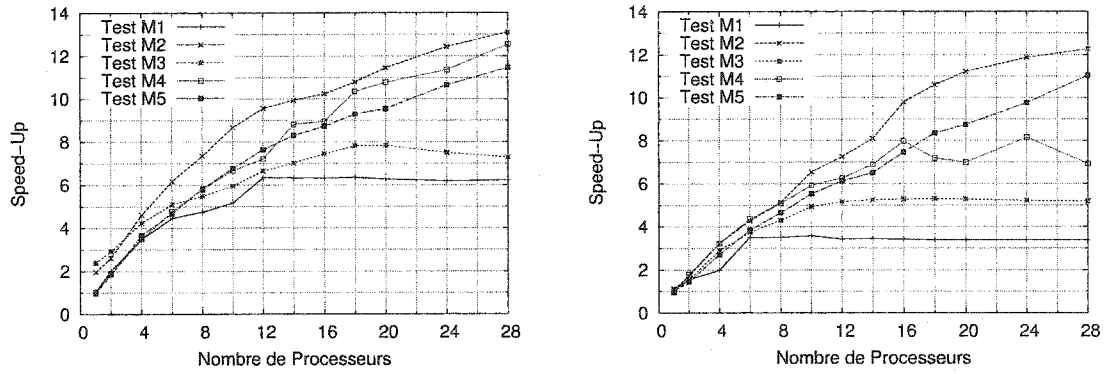
	Temps (sec.) pour chacun des cas tests				
Architecture	M1	M2	M3	M4	M5
HEYSE	93.94	741.32	4017.64	2293.23	3931.51
CHARYBDE	156.55	1197.34	6564.61	3508.11	5661.24

Les figures 4.6 et 4.7 permettent de comparer globalement, pour l'ensemble des cas tests, l'algorithme de partitionnement statique avec l'algorithme de repartitionnement dynamique pour chacune des deux architectures. Clairement, sur HEYSE (figure 4.6), l'algorithme de partitionnement statique est plus performant que l'algorithme de repartitionnement dynamique. Sur CHARYBDE (figure 4.7), bien que la différence entre les deux algorithmes semble moins importante, l'algorithme de partitionnement statique l'emporte également.



(a) Partitionnement statique (b) Repartitionnement dynamique

FIGURE 4.6: Speed-up de IP-**OO**$\mathcal{RT}$ sur HEYSE.

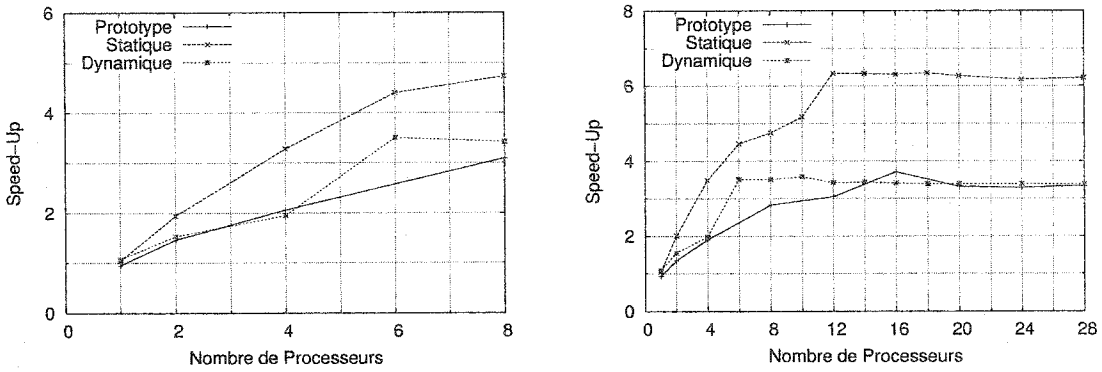


(a) Partitionnement statique (b) Repartitionnement dynamique

FIGURE 4.7: Speed-up de IP-**OO**$\mathcal{RT}$ sur CHARYBDE.

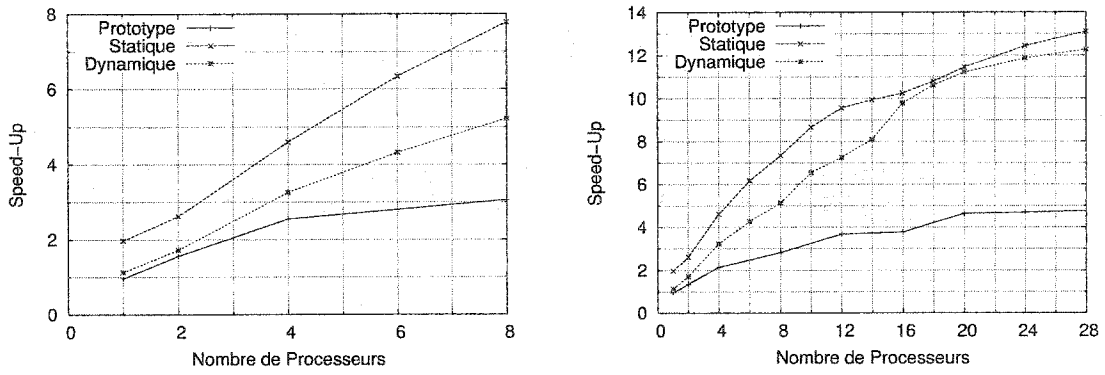
Les cinq figures suivantes (4.8 à 4.12) permettent de comparer les résultats des nouveaux algorithmes avec le prototype. Ces courbes montrent que l'algorithme de partitionnement statique a nettement amélioré les performances de IP-**OO**$\mathcal{RT}$, et ce pour chacun des quatre premiers cas tests. Pour le cinquième, il est tout simplement impossible de faire la comparaison puisque ce cas test n'avait pas été considéré au moment d'évaluer le prototype. Pour ce qui est de l'algorithme de repartitionnement dynamique, ses performances sont aussi meilleures que le prototype mais demeurent inférieures à celles obtenues par le partitionnement statique.

L'objectif premier du nouvel algorithme de partitionnement statique était d'éliminer le



(a) HEYSE (b) CHARYBDE

FIGURE 4.8: Comparaison des speed-up de $\mathbb{P}$-**OO**$\mathcal{RT}$ obtenus avec les nouveaux algorithmes et le prototype pour le cas test M1.

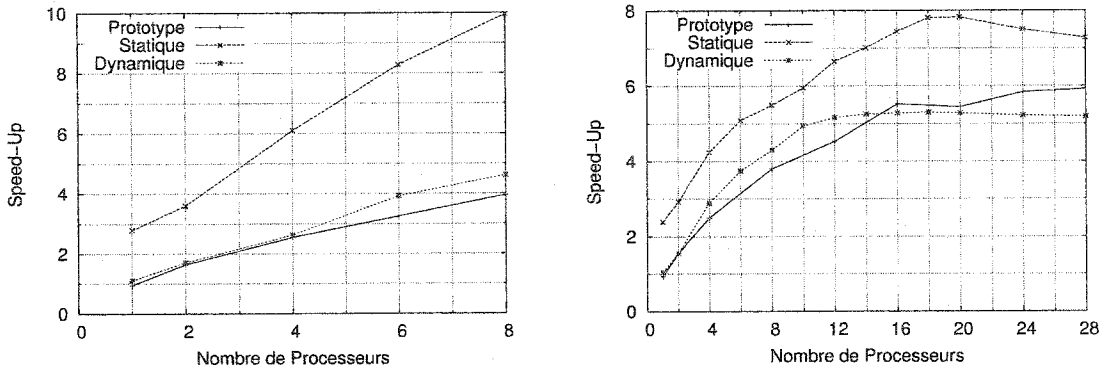


(a) HEYSE (b) CHARYBDE

FIGURE 4.9: Comparaison des speed-up de $\mathbb{P}$-**OO**$\mathcal{RT}$ obtenus avec les nouveaux algorithmes et le prototype pour le cas test M2.

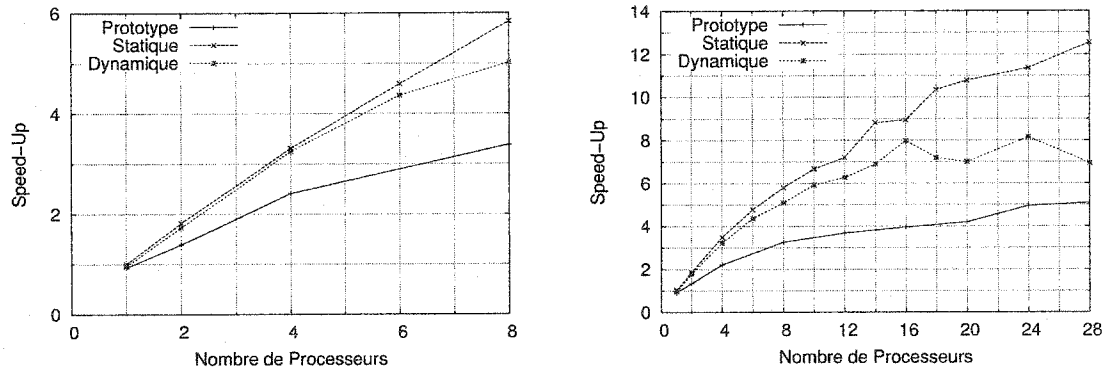
problème d'éléments non conformes. La section traitant de la qualité des maillages obtenus a déjà montré que cet objectif était rempli. Maintenant, les courbes de speed-up obtenues avec ce même algorithme montrent que la performance de l'application parallèle s'est également améliorée de façon notable. En effet, dans la plupart des cas, le speed-up a pratiquement doublé.

Comment expliquer ce phénomène ? Après tout, étant donné que le partitionnement est statique et qu'on ne modifie donc pas la répartition de la charge des processeurs en cours



(a) HEYSE (b) CHARYBDE

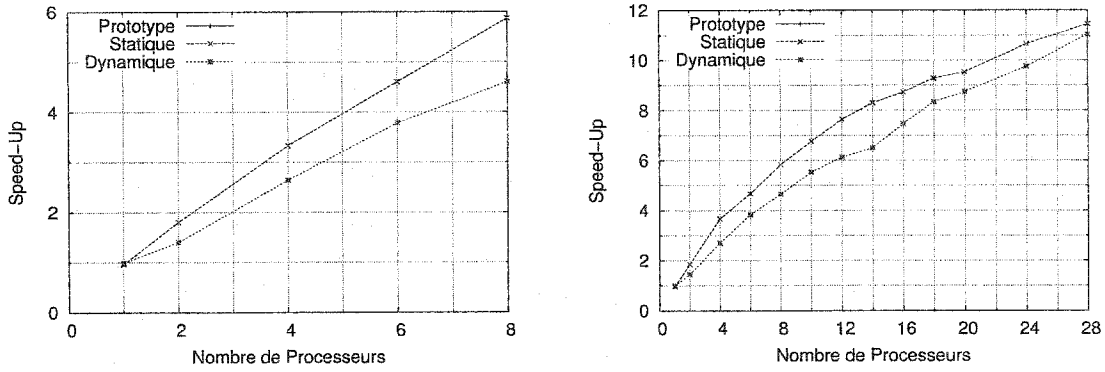
FIGURE 4.10: Comparaison des speed-up de IP-OO$\mathcal{RT}$ obtenus avec les nouveaux algorithmes et le prototype pour le cas test M3.



(a) HEYSE (b) CHARYBDE

FIGURE 4.11: Comparaison des speed-up de IP-OO$\mathcal{RT}$ obtenus avec les nouveaux algorithmes et le prototype pour le cas test M4.

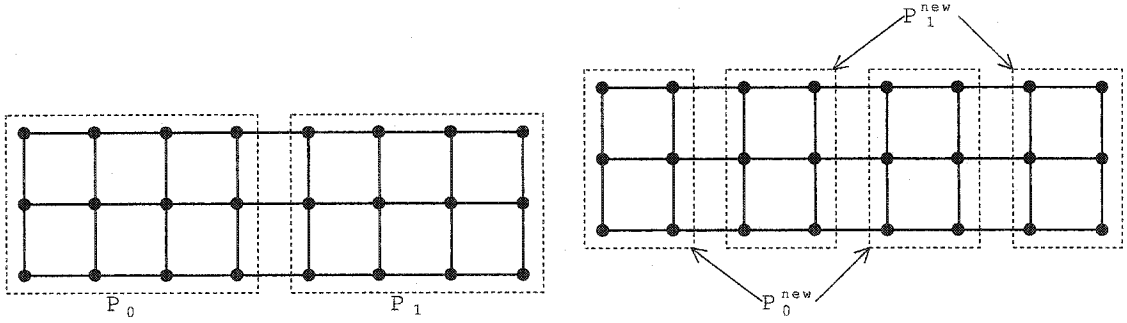
d'exécution, pourquoi le partitionnement statique donne-t-il de meilleurs résultats que le prototype ? En fait, la raison est toute simple. Le partitionnement est statique, soit, mais les partitions générées initialement sont de meilleure qualité. En effet, dans le prototype, chaque processeur possédait une partition composée d'un seul bloc de sommets localisés dans une région particulière du maillage, alors que dans le nouvel algorithme, chaque processeur possède une partition composée de plusieurs blocs plus petits et répartis dans tout le maillage. Cette situation est illustrée à la figure 4.13. Dans le cas du prototype, on obtient les deux partitions P_0 et P_1 tel que présenté à la figure 4.13(a). Dans le



(a) HEYSE (b) CHARYBDE

FIGURE 4.12: Comparaison des speed-up de IP-OO$\mathcal{RT}$ obtenus avec les nouveaux algorithmes pour le cas test M5.

cas du partitionnement statique, des blocs ne possédant aucune couche interne ont été choisis. Chaque bloc est donc constitué de deux couches, soit une frontière gauche et une frontière droite et chaque partition P_0^{new} et P_1^{new} est constituée de plusieurs blocs non liés entre eux tel que présenté à la figure 4.13(b).



(a) Prototype (b) Nouvel algorithme

FIGURE 4.13: Comparaison des partitions obtenues avec le prototype et avec le nouvel algorithme de partitionnement statique.

Ainsi, le principal problème du prototype au point de vue de la performance était que certains sommets devaient être déplacés plus fréquemment que d'autres en procédant par une technique de raffinement du critère local. Or, généralement, la nature même des problèmes d'adaptation fait en sorte que les sommets se déplaçant beaucoup sont regroupés

dans des zones d'intérêt, la plupart du temps près d'un secteur où un phénomène physique intéressant se produit. De cette façon, le processeur qui a la malchance de traiter cette zone (ou ces zones) d'activités importantes aura beaucoup plus de travail que les autres puisque ses sommets seront plus fréquemment déplacés. Dans le nouvel algorithme toutefois, il est possible de faire un partitionnement dans lequel chaque partition est composée de plusieurs blocs plus petits répartis sur tout le maillage. Par exemple, à la figure 4.13, si le problème étudié fait en sorte qu'il y a une activité de déplacement de sommets intense dans la partition P_0 (partition obtenue avec le prototype), le processeur traitant cette partition sera plus chargé que celui qui traite la partition P_1. Cependant, en utilisant le nouvel algorithme, les partitions P_0^{new} et P_1^{new} vont permettre de mieux partager la charge de travail accrue située dans P_0. De façon générale, si les partitions sont composées de petits ensembles de sommets répartis uniformément sur tout le maillage, les probabilités de mieux répartir la surcharge de travail attribuable à ces phénomènes locaux et ainsi d'obtenir des partitions initiales plus équilibrées sont nettement meilleures. Dans cette première série de tests, la taille des blocs était minimale (aucune couche interne dans les blocs). C'est ce qui explique la performance du nouvel algorithme de partitionnement statique. L'influence dans la taille des blocs sera examinée ultérieurement.

Ceci étant dit, qu'en est-il de l'algorithme de repartitionnement dynamique ? Bien qu'il ait augmenté la performance de l'application parallèle comparativement au prototype, l'algorithme de repartitionnement dynamique offre des résultats décevants si on le compare au nouvel algorithme statique. La différence entre les deux algorithmes varie selon les cas tests et les architectures, mais, néanmoins, l'algorithme statique est toujours meilleur. D'abord, il est clair que l'avantage donné par l'algorithme statique d'avoir un partitionnement dont les partitions sont composées de blocs plus petits uniformément répartis sur le domaine est perdu. En effet, dans le cas du repartitionnement dynamique, tel qu'expliqué au chapitre 3, les partitions sont composées d'un seul bloc. Toutefois,

la perte de cet avantage devrait être compensée par le repartitionnement dynamique qui devrait améliorer la qualité des partitions tout au long du processus. Cependant, ce n'est pas le cas. Alors, la première chose à faire de déterminer le temps nécessaire aux phases de repartitionnement dynamique. Si le temps nécessaire est très grand, cela pourrait expliquer que les performances en souffrent puisque le gain obtenu en améliorant les partitions serait annulé par les pertes encourues lors du repartitionnement. Cette question sera examinée plus loin.

4.4.2 Influence du nombre de couches internes

Dans l'algorithme de partitionnement statique, l'usager peut choisir la taille des blocs qui serviront à former les partitions. Pour ce faire, il choisit le nombre de couches internes qui seront attribuées à chaque bloc. À la sous section précédente, il a été vu que plus la taille des blocs est petite, plus la chance d'avoir un partitionnement de bonne qualité est grande. En contre partie, si les blocs sont plus petits, cela signifie qu'il y a moins de couches internes et donc plus de couches frontières. Ayant plus de couches frontières, la communication requise pour la synchronisation des frontières devrait augmenter. Afin de mesurer cet effet, des tests ont été réalisés sur un des cas tests, M4 (Ercoftac), en faisant varier la taille des blocs et en mesurant l'efficacité de l'application. L'efficacité de l'application exécutée sur p processeurs, notée $E(p)$, est calculée à partir du speed-up de l'application pour p processeurs, $S(p)$, à partir de la relation suivante :

$$E(p) = \frac{S(p)}{p} \times 100\%$$

Dans un cas idéal, l'efficacité devrait être constante à 100% peu importe le nombre de processeur utilisé. Toutefois, dans un cas réaliste, l'efficatité devrait graduellement diminuer puisque le volume de communication et la complexité de la synchronisation

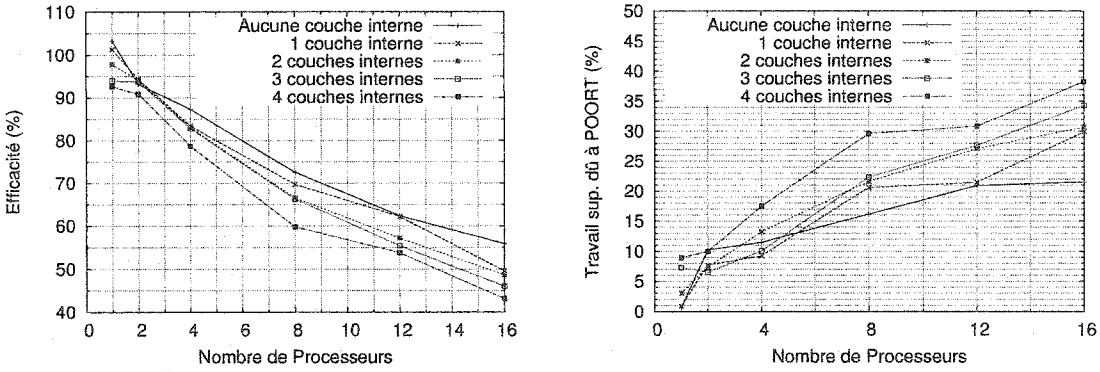
inter processeurs devraient augmenter, entraînant une diminution du speed-up et donc une perte d'efficacité.

La figure 4.14(a) montre les résultats obtenus en terme d'efficacité. On peut y voir que l'efficacité de l'application est meilleure lorsque la taille des blocs est petite. Les meilleurs résultats correspondent aux blocs ne contenant aucune couche interne et les moins bons aux blocs de plus grande taille. Cela montre que la part de communication supplémentaire nécessaire à la gestion des frontières plus nombreuses pour des blocs plus petits est compensée par la qualité des partitions et un meilleur équilibre de charge obtenu avec de petis blocs.

La figure 4.14(b) montre le pourcentage de temps d'exécution ajouté par IP-**OO**$\mathcal{RT}$ afin de partitionner le domaine, communiquer les statistiques, synchroniser les frontières, etc. Ce pourcentage inclut également le temps de synchronisation des processus à la fin d'une itération lorsque tous les processus ne terminent pas en même temps (mauvais équilibre de charge). On y voit que ce pourcentage augmente lorsque la taille des blocs augmente, ce qui confirme que le temps économisé par une plus petite quantité de communication inter itération (les grands blocs ont moins de sommets frontières) est perdu par le temps de synchronisation supplémentaire entre les processeurs dû à un moins bon équilibre de la charge.

4.4.3 Influence de la fréquence de repartitionnement

L'algorithme de repartitionnement dynamique semble donner de moins bons résultats au niveau de la performance de l'application parallèle. C'est ce que montrent les figures 4.8 à 4.12 présentées précédemment. Afin de mieux comprendre les raisons qui expliquent cette mauvaise performance, des tests ont été effectués sur un le cas test M4, ercoftac, en faisant varier la fréquence de repartitionnement. Dans ce test, le repartitionnement a été



(a) Efficacité obtenue en fonction de la taille des blocs

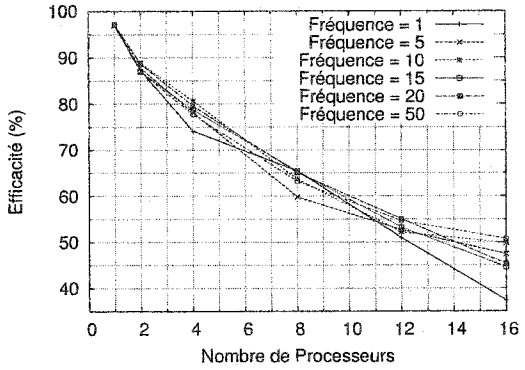
(b) Charge de travail supplémentaire introduite par IP-OO$\mathcal{RT}$

FIGURE 4.14: Influence de la taille des blocs sur la performance de l'algorithme de partitionnement statique pour le cas test M4.

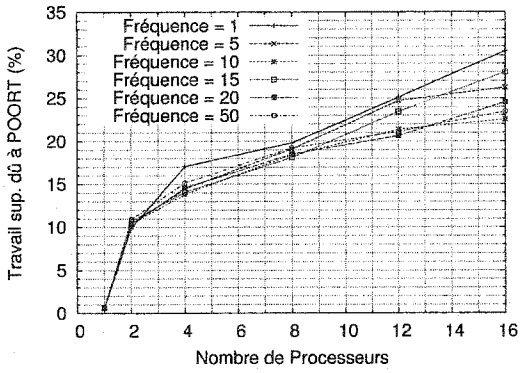
fait par intervalles de 1, 5, 10, 15, 20 et 50 itérations. La figure 4.15(a) montre les courbes d'efficacité obtenues pour chacune de ces fréquences de repartitionnement alors que les figures 4.15(b) et 4.15(c) montrent respectivement le pourcentage de travail ajoutée par IP-OO$\mathcal{RT}$ et la fraction du temps qui est consacrée au repartitionnement.

Comme on peut le voir à la figure 4.15(a), la fréquence de repartitionnement ne semble pas influencer énormément la tendance générale de la courbe d'efficacité. Seule la fréquence de repartitionnement la plus rapide (à toutes les itérations) semble se démarquer des autres, étant significativement plus basse. D'ailleurs, cette tendance peut s'expliquer facilement en examinant la figure 4.15(c). On peut y voir que pour la fréquence de repartitionnement la plus élevée, la proportion de temps passée à faire du repartitionnement est justement beaucoup plus élevée que les autres.

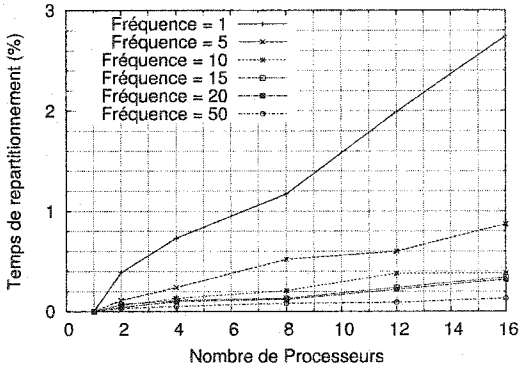
Toutefois, pour les autres fréquences de repartitionnement, les courbes d'efficacité sont très similaires l'une à l'autre et il ne semble pas y avoir de différence significative. D'ailleurs, la courbe qui semble légèrement meilleure que les autres correspond au cas où on repartitionne le moins fréquemment, soit à toutes les 50 itérations, alors que le cas test, avec le critère d'arrêt sélectionné prend généralement à peu près 200 itérations. On



(a) Efficacité obtenue en fonction de la fréquence de repartitionnement



(b) Charge de travail supplémentaire introduite par IP-**OO**$\mathcal{RT}$



(c) Fraction du temps d'exécution pour le repartitionnement

FIGURE 4.15: Influence de la fréquence de repartitionnement.

ne repartitionne donc que 4 fois et cela nous donne un résultat légèrement meilleur que les autres fréquences de repartitionnement. Si le temps de repartitionnement était très élevé, cela s'expliquerait facilement. Or, ce n'est pas le cas ici. La figure 4.15(c) montre que le temps de repartitionnement, si on fait exception du cas extrême où le repartitionnement est fait à toutes les itérations, est inférieur à 1% du temps d'exécution global et même inférieur à 0.4%, pour les fréquences supérieures à 5. Le temps de repartitionnement ne devrait donc pas influencer fortement le temps d'exécution du système pour des fréquences de repartitionnement raisonnables (supérieures à 5). La figure 4.15(b) montre également qu'il n'y a pas de différence majeure dans la charge de travail ajoutée par IP-**OO**$\mathcal{RT}$.

En résumé, la figure 4.15 semble montrer que la fréquence de repartitionnement n'a pas vraiment d'influence majeure sur le temps d'exécution, sauf pour le cas extrême de repartitionnement à toutes les itérations. De plus, il a été vu que le repartitionnement dynamique donne de moins bons résultats que le partitionnement statique et ce malgré un temps de repartitionnement négligeable par rapport au reste. Ces différents indices semblent plutôt révéler d'autres problèmes.

Premièrement, un problème semble être lié à la fonction de poids associée à chaque sommet. La fonction de poids utilisée, telle que présentée au chapitre 3 consiste, pour chaque sommet, à garder un historique de ses déplacements en utilisant une formule de sous-relaxation de la forme $P_s^k = \alpha P_s^{k-1} + (1-\alpha)D_s^k$. Normalement, plus la valeur contenue dans cet historique est élevée, plus le sommet a de chances d'être déplacés dans les itérations subséquentes. Ainsi, l'algorithme de repartitionnement tente d'équi-distribuer les poids des sommets de sorte que chaque partition ait le même poids total. Cependant, la décision de déplacer ou non un sommet durant une itération dépend uniquement du fait que la valeur de son poids est supérieure ou inférieure au critère local. Ainsi, même si deux sommets doivent être déplacés au cours d'une itération (poids supérieur au critère local), si un des sommets a un poids beaucoup plus élevé que l'autre, il comptera comme un sommet nécessitant une charge de travail plus grande, même si, dans les faits, les sommets nécessiteront le même travail (la charge de travail requise pour déplacer un sommet ne dépend pas de la magnitude du déplacement). Cependant, si on remplace cet historique par une valeur booléenne indiquant simplement si le sommet devra être déplacé au cours de l'itération suivante, la prédiction du travail à faire sur le sommet à plus long terme est perdue. Il n'est donc pas très évident de trouver une bonne fonction de poids pour les sommets dans le cadre du repartitionnement.

Il existe également un autre problème fondamental qui explique la difficulté de repartitionner efficacement. Il s'agit de la granularité du partitionnement. Afin de préserver la

validité des éléments, il est nécessaire de partitionner par couches. Ainsi, une partition ne peut être formée que d'un nombre entier de couches, ce qui limite la flexibilité dans l'attribution des sommets. Par exemple, dans le cas test ercoftac (M4), il y a 127982 sommets, mais seulement 104 couches de sommets. Il est beaucoup plus difficile d'équilibrer la charge en répartissant les sommets par couches qu'en répartissant les sommets individuellement. En outre, les partitions doivent être composées d'au moins deux couches chacune (les deux frontières) et toutes les couches d'une partition doivent être voisines l'une de l'autre. Ces différentes contraintes empêchent d'avoir un bon contrôle sur la répartition du poids des sommets.

En résumé, il est clair que l'algorithme de partitionnement basé sur la construction de couches de sommets se prête mal au repartitionnement dynamique en raison des contraintes de construction qui ne permettent pas suffisamment de flexibilité dans la formation des partitions. De plus, il semble assez difficile de concevoir une fonction de poids efficace sur laquelle baser le repartitionnement dynamique.

4.4.4 Ratio entre le temps de calcul et le temps de communication

Dans une application parallèle, typiquement, il y a deux variables à observer au niveau du temps d'exécution. Généralement, le temps parallèle de l'application sur p processeurs, noté $t(p)$ est exprimé sous la forme suivante :

$$t(p) = t_{\text{calcul}} + t_{\text{communication}}$$

où t_{calcul} est le temps passé à faire du travail de calcul et $t_{\text{communication}}$ est le temps requis pour la synchronisation et la communication entre les processeurs. En supposant que la répartition de la charge de travail entre les processeurs est parfaite, c'est-à-dire que chaque processeur a une somme de travail identique, t_{calcul} peut s'exprimer comme

étant :

$$t_{\text{calcul}} = \frac{t_{\text{séquentiel}}}{p}$$

où $t_{\text{séquentiel}}$ est le temps d'exécution de l'application séquentielle.

Le tableau 4.5 montre le temps de calcul et le temps de communication pour le cas test M4 en mode de partitionnement statique. On y présente le temps de calcul du processeur ayant la plus grosse charge de travail, celui du processeur ayant la plus petite charge de travail et la moyenne. Cela permet d'avoir une idée de l'équilibre de la charge. Pour ce qui est du temps de communication, la variation entre le temps maximal et le temps minimal représente le temps d'attente, c'est-à-dire le temps pendant lequel le processeur le moins occupé devait attendre que le processeur le plus occupé ait terminé son travail avant d'entreprendre les communications inter itérations pour la communication des statistiques et la synchronisation des frontières. Le temps minimal correspond donc au temps vraiment passé à faire des communications et la différence avec le temps maximal correspond au temps d'attente dû à un mauvais équilibre de la charge. Finalement, le tableau 4.5 présente également le ratio $t_{\text{calcul}}/t_{\text{communication}}$ à partir des valeurs moyennes. Il s'agit d'une bonne indication de l'efficacité de l'application parallèle.

TABLEAU 4.5 Comparaison du temps de calcul et du temps de communication dans IP-OORT pour le cas test M4.

Nbre de proc.	t_{calcul} Min.	t_{calcul} Max.	t_{calcul} Moy.	$t_{\text{communication}}$ Min.	$t_{\text{communication}}$ Max.	$t_{\text{communication}}$ Moy.	Ratio
1	3302.0	3302.0	3302.0	22.1	22.1	22.1	149.4
2	1580.8	1631.6	1606.2	176.6	207.8	192.2	8.4
4	760.6	807.9	794.7	96.0	143.4	115.4	7.0
8	354.8	440.2	400.8	54.7	140.1	97.7	4.1
12	236.6	370.0	278.2	85.4	183.6	145.0	1.9
16	178.4	248.8	200.9	34.8	105.3	84.5	2.4

Normalement, plus le ratio est élevé, plus la performance de l'application parallèle devrait être bonne. Dans le tableau 4.5, on observe que le ratio du temps de calcul sur le

temps de communication diminue rapidement. Le tableau 4.6 donne quelques indices supplémentaires quant à la raison de cette dégradation. On y présente séparément le temps de communication des statistiques et le temps de synchronisation des frontières. On remarque immédiatement qu'au fur et à mesure que le nombre de processeur augmente, le temps de communication des statistiques augmente substantiellement alors que le temps de synchronisation des frontières diminue. Si le temps de synchronisation des frontières diminue, c'est tout simplement parce que les frontières sont plus petites (chaque processeur a moins de sommets frontières) et que les processeurs s'envoient leurs frontières deux à deux simultanément. Toutefois, pour ce qui est du temps de communication des statistiques, il est anormal qu'il augmente ainsi (après tout, ce n'est qu'un `MPI_Allgather()` avec une quantité de données restreinte). Toutefois, cette augmentation peut s'expliquer par le fait que le temps de communication des statistiques inclut le temps d'attente en fin d'itération lorsque les processeurs ne terminent pas leur travail en même temps puisque la communication collective associée à la communication des statistiques constitue un point de synchronisation à la fin d'une itération. D'ailleurs, si on prend le cas à 12 processeurs, on réalise que bien que le temps moyen de communication soit de 111.8, le temps minimal est de 16.1 et le temps maximal est de 148.3. Cela signifie que le processeur le moins occupé a dû attendre 132.2 secondes au total durant l'adaptation que le processeur le plus occupé ait terminé son travail. Le temps de communication comme tel est beaucoup plus raisonnable. Il s'agit donc encore une fois d'un problème d'équilibre de charge. En outre, on remarque que les cas où le ratio est le plus faible coïncident avec les cas où le temps de communication des statistiques est le plus élevé. Il s'agit des cas où l'équilibre de la charge est le moins bon.

En résumé, le ratio du temps de calcul sur le temps de communication dans IP-OORT semble mauvais, mais en réalité, le temps des communication est tout à fait raisonnable, c'est plutôt le temps d'attente dû à un mauvais équilibre de charge qui affecte les performances. En effet, bien que le problème ait été amélioré depuis le prototype, les données

TABLEAU 4.6 Comparaison du temps de calcul et du temps de communication dans IP-OO$\mathcal{RT}$ pour le cas test M4.

Nbre de proc.	Ratio	Temps de comm. des stats	Temps de synchr.
1	149.4	0.0	0.0
2	8.4	22.2	158.3
4	7.0	28.3	79.6
8	4.1	52.5	40.6
12	1.9	111.8	29.1
16	2.4	60.2	21.5

des tableaux 4.5 et 4.6 montrent qu'il y a encore place à de l'amélioration à ce niveau.

4.5 Discussion

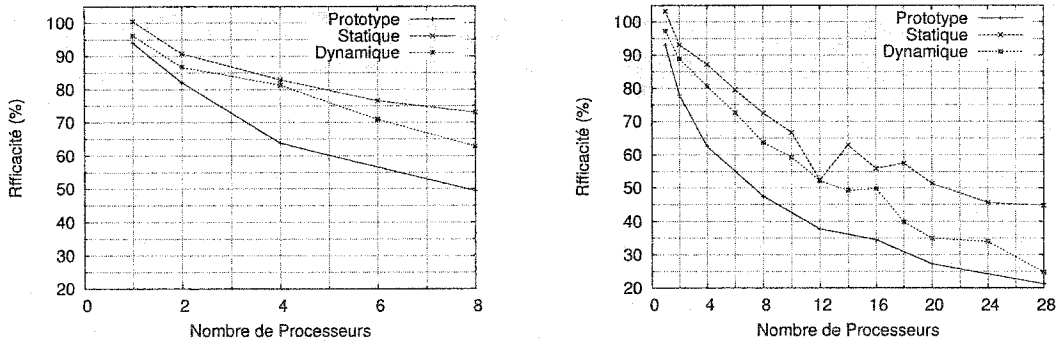
Les résultats obtenus avec le nouvel algorithme de partitionnement semblent prometteurs. Toutefois, certains problèmes subsistent, particulièrement dans le cas du repartitionnement dynamique. La présente section se veut donc une analyse des principales forces et faiblesses de la nouvelle version parallèle. La possibilité d'étendre l'adaptation parallèle aux maillages non structurés sera également analysée.

4.5.1 Forces de IP-OO$\mathcal{RT}$

Au chapitre des points forts de IP-OO$\mathcal{RT}$, on remarque d'abord que des gains importants ont été réalisés dans la nouvelle version par rapport au prototype. D'abord, le problème d'éléments non conformes a été réglé, ce qui était capital. En effet, il aurait été complètement inutile d'avoir une version parallèle extrêmement performante au point de vue de la vitesse d'exécution, mais qui génère des maillages invalides et donc inutilisables. L'algorithme développé assure que deux sommets voisins situés sur deux partitions distinctes ne seront pas déplacés simultanément, ce qui assure la cohésion des

frontières et empêche les déplacements conflictuels qui étaient responsables de la création de ces éléments non conformes.

En outre, le nouveau partitionnement obtenu a permis d'améliorer la performance de l'application parallèle de façon importante. Le speed-up pour les différents cas tests ont pratiquement doublé. Le nouveau partitionnement permet de mieux répartir la charge de travail parmi les processeurs par le simple fait que les partitions de chaque processeur sont composées de blocs de taille relativement petite et uniformément répartis sur le domaine. Ainsi, la surcharge de travail associée à une certaine zone du maillage où un phénomène quelconque génère une forte activité peut être mieux répartie. La figure 4.16 résume assez bien la situation en présentant une comparaison du prototype avec les deux nouveaux algorithmes (partitionnement statique et repartitionnement dynamique) pour le cas test Ercoftac. On peut remarquer que le nouvel algorithme de partitionnement statique a nettement amélioré l'efficacité de l'application parallèle.



(a) HEYSE (b) CHARYBDE

FIGURE 4.16: Comparaison des courbes d'efficacité du prototype, de l'algorithme de partitionnement statique et l'algorithme de partitionnement dynamique pour le cas test Ercoftac (M4).

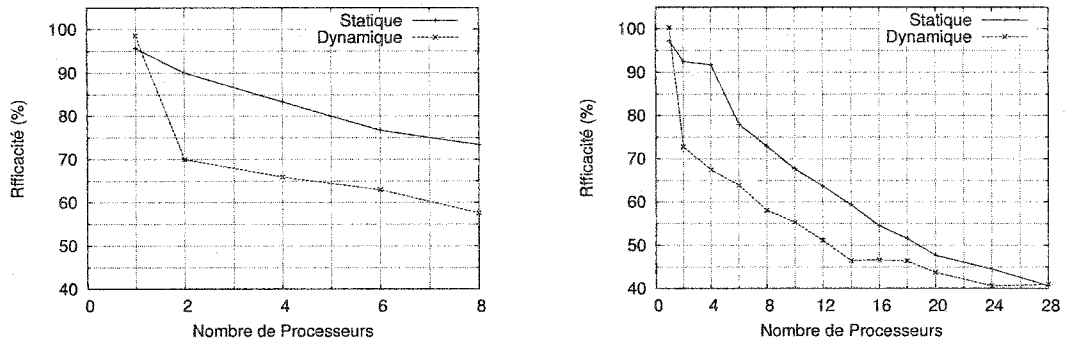
En ce qui a trait à la qualité des maillages obtenus, il a été montré que $\mathbb{P}$-**OO**$\mathcal{RT}$ donne des maillages équivalents à **OO**$\mathcal{RT}$. En effet, aucune différence majeure n'a été notée quant à la longueur moyenne des arêtes dans l'espace métrique ou à l'écart type de cette longueur. Dans plusieurs cas, une légère amélioration (longueur plus près de la

valeur unitaire et écart type plus faible) a même été remarquée. Il s'agit d'une excellente nouvelle, car il est important que les résultats obtenus par $\mathbb{P}$-**OO**$\mathcal{RT}$ soient de qualité équivalente à ceux produits par **OO**$\mathcal{RT}$, sinon, l'application parallèle ne serait pas très utile.

4.5.2 Problèmes identifiés

Malgré ces résultats encourageants, plusieurs problèmes ont été identifiés dans $\mathbb{P}$-**OO**$\mathcal{RT}$. D'abord, bien que la performance de l'application ait été beaucoup améliorée par rapport au prototype, les résultats obtenus montrent qu'il serait possible de faire beaucoup mieux, car le problème d'équilibre de charge, bien que de moindre importance, est toujours présent. À ce niveau, l'algorithme de repartitionnement dynamique aurait dû permettre d'améliorer la répartition de la charge mais, ce n'est pas le cas. En effet, l'algorithme de repartitionnement dynamique offre de moins bonnes performances que l'algorithme de partitionnement statique. La figure 4.16, présentée à la sous-section précédente, l'illustre bien. En effet, bien que l'algorithme de repartitionnement dynamique donne de meilleurs résultats que le prototype, ceux-ci restent inférieurs à ceux obtenus avec le partitionnement statique. La figure 4.17, qui présente une comparaison entre les algorithmes statique et dynamique pour le cas test Dt151_Sweden, illustre la même situation.

Plusieurs raisons expliquent la piètre performance de l'algorithme de repartitionnement dynamique. Premièrement, la difficulté de maintenir une fonction de poids qui représente correctement le travail à faire sur un sommet cause un problème majeur. En effet, comment repartitionner les sommets de façon à équidistribuer le poids si la fonction de poids elle-même ne constitue pas une bonne mesure de la charge de travail associée à un sommet ?



(a) HEYSE (b) CHARYBDE

FIGURE 4.17: Comparaison des courbes d'efficacité de l'algorithme de partitionnement statique et de l'algorithme de partitionnement dynamique pour le cas test Dt151_Sweden (M5).

Deuxièmement, la granularité du partitionnement obtenu est trop faible. En effet, on ne peut pas répartir les sommets individuellement, on doit absolument les répartir en attribuant une couche complète à la fois. De plus, chaque partition doit être composée d'un minimum de deux couches voisines et toutes ses couches doivent être voisines l'une de l'autre dans le domaine. Ceci restreint les possibilités de distribution uniforme du poids sur les processeurs. Par exemple, si une couche comporte plusieurs sommets ayant un poids important, il est impossible de la diviser pour obtenir un partitionnement distribuant les poids plus équitablement. Ainsi, dans le cas test M4, il y a 127982 sommets répartis sur 104 couches. Chaque couche possède alors en moyenne 1230 sommets. De plus, pour des maillages multi-blocs, comme M4 et M5, il est tout à fait possible, et même probable que les couches n'aient pas toutes le même nombre de sommets, ce qui complique la tâche de création de partitions à poids uniforme.

4.5.3 Maillages non structurés

$\mathbb{P}$-**OO**$\mathcal{RT}$ fonctionne uniquement sur les maillages structurés. Maintenant, serait-il envisageable d'ajouter le support pour les maillages non structurés ? L'adaptation de

maillages non structurés comprend plusieurs opérations modifiant la connectivité du maillage comme le raffinement, le déraffinement et le retournement. La modification de la connectivité des éléments du maillage est beaucoup plus complexe en parallèle. En outre, l'architecture actuelle de $\mathbb{P}$-**OO**$\mathcal{RT}$, où chaque processeur possède une copie complète du maillage ne se prête pas bien à ces opérations. En effet, à chaque fois qu'un sommet est créé sur une partition, il faudrait que tous les processeurs soient mis au courant afin d'ajouter ce sommet dans leur maillage global en transmettant toutes les informations requises à la reconstruction de la connectivité.

Dans un premier temps, l'option de permettre uniquement le déplacement des sommets dans $\mathbb{P}$-**OO**$\mathcal{RT}$ pour les maillages non structurés a été analysée. La figure 4.18 montre l'algorithme d'adaptation en non structuré de **OO**$\mathcal{RT}$ à haut niveau. L'algorithme est constitué d'une grande boucle principale dans laquelle chacune des techniques d'adaptation est appelée dans un ordre séquentiel précis. On divise les opérations en deux types. D'abord, les opérations de retournement des arêtes et faces et les opérations de déplacements de sommets sont considérées comme des opérations de lissage du maillage, car elles n'en changent pas la densité comparativement aux opérations de raffinement et de déraffinement. Ainsi, dans chacun des tours de la boucle principale, une première phase de lissage est lancée (incluant une phase de déplacement de sommets). Ensuite, l'opération de raffinement est lancée. Puis, on lance de nouveau une opération de lissage avant de déraffiner les arêtes. On termine ensuite sur une dernière opération de lissage. Les opérations de lissage permettent d'améliorer la qualité des éléments qui peut avoir été altérée par les opérations de raffinement et de déraffinement. Chacune des opérations de lissage inclut une étape de déplacement des sommets. En outre, on sait que c'est l'opération de déplacement des sommets qui est la plus coûteuse dans l'adaptation en non structuré. Donc, si on arrive à paralléliser cette étape, l'adaptation en non structuré devrait en bénéficier.

```
tant que Le maillage n'est pas convergé faire
    début Bloc Lissage
        Phase de retournement d'arêtes;
        Phase de retournement de faces;
        Phase de déplacement de sommets;
    fin
    Phase de raffinement d'arêtes;
    début Bloc Lissage
        Phase de retournement d'arêtes;
        Phase de retournement de faces;
        Phase de déplacement de sommets;
    fin
    Phase de déraffinement d'arêtes;
    début Bloc Lissage
        Phase de retournement d'arêtes;
        Phase de retournement de faces;
        Phase de déplacement de sommets;
    fin
fin
```

FIGURE 4.18: Algorithme de haut niveau de l'adaptation en non structuré.

À prime abord, il n'y a pas de difficulté majeure à paralléliser le déplacement des sommets en non structuré puisqu'il s'agit en fait de la même fonction qui est appelée pour le déplacement des sommets que dans le cas des maillages structurés (`deplacer_lesSommets()`). Or, puisque les autres techniques d'adaptation ne sont pas parallélisées, seul un des processeurs devra les exécuter et il faudra, avant chaque opération de déplacement que ce processeur envoie aux autres le nouveau maillage courant. Il ne s'agit plus ici de seulement indiquer la nouvelle position des sommets. En effet, des sommets ont été créés, d'autres ont été supprimés et la connectivité des éléments a changé. Comme l'algorithme de déplacement des sommets requiert une connaissance complète de tout le maillage par tous les processeurs, il faut, avant chaque opération de déplacement, que le processeur ayant fait les autres opérations transmette le nouveau maillage courant aux autres. À la fin du déplacement, il faudra également mettre à jour la position de tous les sommets sur le processeur maître, chargé de faire les autres opérations par la suite.

Normalement, cela ne constitue pas un problème, car la phase de finalisation n'est faite qu'une fois à la fin du déplacement et prend en moyenne moins de 1% du temps total d'exécution. Cependant, en non structuré, la phase de déplacement est appelée plus fréquemment (généralement pour un nombre d'itérations restreint à chaque fois) et la phase de finalisation sera nécessaire à chaque fois ce qui risque de faire diminuer la performance très rapidement. En ajoutant la nécessité de communiquer pour reconstruire le maillage sur tous les processeurs avant la phase de déplacement, on obtiendra un algorithme très peu performant. Ces différentes raisons expliquent pourquoi le support des maillages non structurés a été laissé de côté. Pour le supporter, il faudrait revoir en entier les algorithmes de IP-OO$\mathcal{RT}$ et concevoir, entre autres, une technique de partitionnement qui ne nécessitent pas la connaissance de tout le maillage pour chacun des processeurs.

CONCLUSION

Dans un schéma de simulation numérique, il est nécessaire d'avoir un contrôle sur l'erreur. Pour contrôler l'erreur, il est nécessaire de faire de l'adaptation sur les maillages. Cela permet de diminuer l'erreur. En effet, le problème à résoudre est non linéaire. Le maillage influence la solution et la solution influence le maillage. En utilisant le processus d'adaptation de maillage dans le cadre de ce schéma, éventuellement, le problème convergera vers une solution optimale et vers un maillage correspondant à cette solution. L'étape d'adaptation est partie intégrante du schéma de simulation. Or, plusieurs travaux de parallélisation ont été faits dans les autres étapes du schéma global comme la génération de maillage et la résolution du problème (par éléments finis, volumes finis, etc.), mais très peu ont été faits dans le cadre de l'adaptation de maillage. Puisque l'adaptation s'inscrit dans le schéma global, il était important d'en améliorer les performances afin que celle-ci ne devienne pas un goulot d'étranglement dans le processus de simulation.

C'est pour répondre à ce besoin d'accélération du processus d'adaptation que IP-**OO**$\mathcal{RT}$ est né. IP-**OO**$\mathcal{RT}$ parallélise les algorithmes de **OO**$\mathcal{RT}$, une bibliothèque pour l'adaptation de maillage. Dans un premier temps, un prototype a été développé. Ce prototype ne supporte que les maillages structurés et la librairie ParMETIS était utilisé pour le partitionnement du domaine. Plusieurs lacunes ont été observées, notamment en ce qui concerne la génération d'éléments non conformes due à une mauvaise gestion des sommets situés aux frontières entre les partitions. En outre, l'accélération obtenue en parallèle était très décevante.

Afin de résoudre ce problème, de nouveaux algorithmes ont été conçus et implantés. Ces algorithmes sont basés sur la construction de couches de sommets qui respectent différentes contraintes assurant qu'un sommet situé sur une couche ne peut avoir pour

voisin que des sommets de la même couche et des deux couches voisines immédiates. Il s'agit en fait d'un algorithme de coloriage de graphe.

Il est alors possible de déplacer uniquement une moitié des sommets à chaque itération de sorte qu'un sommet ne sera pas déplacé au cours d'une itération si un de ses voisins situés sur une autre partition l'est. De cette façon, les problèmes d'éléments non conformes ont été réglés. De plus, la qualité globale des maillages obtenus avec $\mathbb{P}$-**OO**$\mathcal{RT}$ a été démontrée équivalente à celle de maillages obtenus avec **OO**$\mathcal{RT}$. Maintenant, au niveau de la performance, les nouveaux algorithmes de partitionnement ont également permis d'améliorer substantiellement l'accélération parallèle. Toutefois, l'algorithme de repartitionnement dynamique, qui devait permettre d'augmenter encore plus la performance, n'offre pas encore de très bons résultats. Cela est principalement dû aux problèmes d'identification d'une fonction de poids représentative pour les sommets et à la faible granularité du partitionnement.

Travaux à faire dans le futur

À la lumière de ces résultats, plusieurs conclusions peuvent être tirées quant aux directions à prendre pour les développements futurs de $\mathbb{P}$-**OO**$\mathcal{RT}$. Premièrement, bien que les nouveaux algorithmes de partitionnement aient amélioré la situation, l'équilibre de la charge de travail entre les processeurs demeure encore relativement mauvais. Afin de résoudre ce problème, la première priorité devrait être de concevoir et implanter une fonction de poids associée à chaque sommet qui représente plus fidèlement la charge de travail associée à ce sommet. Il s'agit là d'un travail assez difficile à résoudre en utilisant les techniques actuellement connues en adaptation de maillage, car il est difficile, même si on connaît l'historique de déplacement d'un sommet de prédire s'il sera déplacé dans les itérations suivantes (on ne peut connaître avec certitude la réponse que pour la

prochaine itération, mais pas les autres).

Le second point à examiner devrait être la granularité du partitionnement. Il serait très intéressant pour l'algorithme de repartitionnement de permettre une granularité plus fine. Serait-il possible de diviser les couches d'une certaine façon tout en assurant toujours la cohérence des frontières ? C'est une avenue qui mérite d'être explorée, car une granularité plus fine pourrait permettre d'obtenir un meilleur équilibre de la charge.

Le troisième point à considérer concerne le partitionnement lui-même. Puisque chaque processeur possède une copie complète du maillage, il est possible, lors du déplacement des sommets en structuré, d'avoir des algorithmes très performants qui ne s'échangent qu'un minimum d'information. Cela limite beaucoup le niveau de communication. Cependant, lorsque vient le temps d'introduire l'adaptation en non structuré, ne serait-ce que pour le déplacement des sommets, la nécessité de connaître tout le maillage par tous les processeurs devient difficile à gérer. Il serait alors très intéressant de voir les possibilités de modifier l'algorithme de partitionnement de façon à pouvoir prendre en compte les modifications locales de raffinement, déraffinement et de retournement sans avoir à les transmettre à tous les processeurs. Ceci rendrait plus facile la parallélisation du processus de déplacement des sommets en non structuré, car chaque processeur fonctionnerait de façon plus indépendante.

Malheureusement, la phase de finalisation du déplacement des sommets en parallèle devra toujours être appelée à la fin du déplacement pour que le processeur chargé de faire les opérations séquentielles de raffinement, déraffinement et de retournement puisse les faire sur le maillage mis à jour. Comme la fonction de déplacement est appelée souvent en non structuré pour faire un nombre restreint d'itérations à chaque fois, cela pourrait dégrader les performances. Il semble donc que pour paralléliser l'adaptation en non structuré, on ne peut pas vraiment se contenter de paralléliser une seule des étapes, en l'occurrence le déplacement des sommets, il faudrait plutôt paralléliser toutes les étapes

et développer un nouvel algorithme de partitionnement qui s'y prête bien. Plusieurs algorithmes de partitionnement existent dans la littérature, mais peu de gens les ont intégrés dans un processus d'adaptation de maillage regroupant des opérations de déplacement de sommets, de raffinement, de déraffinement et de retournement. Le développement de nouveaux algorithmes de partitionnement et la parallélisation de toutes les étapes d'adaptation représentent, à mon avis, la voie à suivre pour l'intégration des maillages non structurés à $\mathbb{P}$-**OO**$\mathcal{RT}$.

RÉFÉRENCES

ANNAMALAI, V., KRISHNAMOORTHY, C. S. ET KAMAKOTI, V. (1999). High-speed applications in the automotive industry : adaptive finite element analysis on a parallel and distributed environment. *Parallel Computing*, 25, 1413–1434.

COWEN, L. J. ET JESURUM, C. E. (1997). Coloring with defect. *SODA : ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

DOMPIERRE, J. ET LABBÉ, P. (1999). *OORT (Object-Oriented Remeshing Toolkit), Manuel de l'usager*. CERCA, Montréal, Québec.

FREITAG, L., JONES, M. ET PLASSMANN, P. (1999). A parallel algorithm for mesh smoothing. *SIJSSC : SIAM Journal on Scientific and Statistical Computing, apparently renamed SIAM Journal on Scientific Computing*, 20.

FREY, P. J. ET GEORGE, P.-L. (1999). *Maillages. Applications aux éléments finis*. Hermès, Paris.

GAMMA, E., HELM, R., JOHNSON, R. ET VLISSIDES, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley.

GEORGE, P.-L., éditeur (2001). *Maillage et adaptation*. Hermès, Paris.

GUATTERY, S. ET MILLER, G. L. (1995). On the performance of spectral graph partitioning methods. *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 233–242.

GUIBAULT, F., GARON, A., OZELL, B. ET CAMARERO, R. (1995). Analysis and visualization tools in CFD. Part II : A case study in grid adaptivity. *Finite Elements in Analysis and Design*, 19, 309–324.

GUIBAULT, F., VU, T. ET CAMARERO, R. (1999). Automatic blocking for hybrid grid generation in hydraulics components. *International Journal on Hydropower and Dams*, 5, 81–86.

HENDRICKSON, B. ET LELAND, R. W. (1995). A multi-level algorithm for partitioning graphs. *Supercomputing*.

KARGER, D. R., MOTWANI, R. ET SUDAN, M. (1994). Approximate graph coloring by semidefinite programming. *IEEE Symposium on Foundations of Computer Science*. 2–13.

KARYPIS, G. ET KUMAR, V. (1995). A fast and high quality multilevel scheme for partitioning irregular graphs. Rapport technique TR 95-035.

KARYPIS, G. ET KUMAR, V. (1998a). *METIS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. University of Minnesota, Department of Computer Science and Army HPC Research Center, Minneapolis, MN.

KARYPIS, G. ET KUMAR, V. (1998b). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48, 96–129.

KARYPIS, G., SCHLOEGEL, K. ET KUMAR, V. (2002). *PARMETIS, Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.0*. University of Minnesota, Department of Computer Science and Army HPC Research Center, Minneapolis, MN.

KORNERUP, J. (1997). Odd-even sort in powerlists. *Information Processing Letters*, 61, 15–24.

LABBÉ, P., GUIBAULT, F. ET CAMARERO, R. (2000). Core classes for a multi-disciplinary numerical analysis framework. *7th International Conference on Numerical Grid Generation in Computational Field Simulations*. Whistler, B.-C., 933–942.

LABBÉ, P., GUIBAULT, F., DOMPIERRE, J., VALLET, M.-G. ET TRÉPANIER, J.-Y. (2001). A generic mesher for STEP compliant geometries. *Proceedings of the 48th Annual CASI Conference*. Canadian Aeronautics and Space Institute, Toronto, Ontario, 77–83.

MORIN, J.-P. (2003). Partitionnement à l'aide de ParMETIS pour POORT. Rapport technique.

OLIKER, L. ET BISWAS, R. (2000). Parallelization of a dynamic unstructured algorithm using three leading programming paradigms. *IEEE Transactions on Parallel and Distributed Systems*, 11, 931–940.

OZELL, B., CAMARERO, R., GARON, A. ET GUIBAULT, F. (1995). Analysis and visualization tools in CFD. Part I : A configurable data extraction environment. *Finite Elements in Analysis and Design.*

POMBO, J. J., CABALEIRO, J. C. ET PENA, T. F. (2001). Parallel complete remeshing for adaptive schemes. *International Conference on Parallel Processing Workshops*. 73–78.

POTHEN, A., SIMON, H. D. ET LIOU, K.-P. (1990). Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 430–452.

SELWOOD, P. M. ET BERZINS, M. (1999). Parallel unstructured tetrahedral mesh adaptation : algorithms, implementation and scalability. *Concurrency : Practice and Experience*, 11, 863–884.

SIMON, H. D. (1991). Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2, 135–148.

SIROIS, Y., DOMPIERRE, J., VALLET, M.-G., LABBÉ, P. ET GUIBAULT, F. (2002). Progress on vertex relocation schemes for structured grids in a metric space. *8th International Conference on Numerical Grid Generation*. Honolulu, Hawaii.

SOHN, A., BISWAS, R. ET SIMON, H. D. (1996). A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors. *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*. 189–192.

VU, T. C., GUIBAULT, F., DOMPIERRE, J., LABBÉ, P. ET CAMARERO, R. (2000). Computation of fluid flow in a model draft tube using mesh adaptive techniques. *Proceedings of the Hydraulic Machinery & Systems 20th IAHR Symposium.*

ZHOU, X. ET NISHIZEKI, T. (2000). Graphs coloring algorithms. *IEICE Trans. on Information and Systems*, E83-D, 407–418.

ANNEXE I

Manuel d'utilisation de OO$\mathcal{RT}$

Cet annexe ne vise pas à présenter un manuel d'utilisation complet de OO$\mathcal{RT}$. OO$\mathcal{RT}$ possède de multiples options et supporte plusieurs type de maillage. Toutefois, dans le cadre de ce projet, seul le déplacement de sommets dans les maillages structurés est utilisé, alors la présentation sera centrée sur ce point.

I.1 Ligne de commande

Voyons d'abord un exemple de ligne de commande qui correspond à la ligne de commande utilisée pour le cas test M1 :

```
oort.exe -H var -c oort.cfg -o oort.pie arctg_hex.pie}
```

L'option « -H » permet de spécifier le nom du champs contenant la solution à partir de laquelle la métrique utilisée pour l'adaptation sera construite. Cela doit correspondre à un champs défini dans le fichier d'entrée. Dans ce cas-ci, le nom du champs est « var ».

L'option « -c » permet de spécifier un fichier contenant la configuration à utiliser pour l'adaptation. Dans ce cas-ci, le nom du fichier de configuration est « oort.cfg ». Une explication sur le contenu de ce fichier de configuration sera faite à la section suivante.

L'option « -o » permet de spécifier un fichier de sortie. Ce fichier contiendra le maillage adapté produit par OO$\mathcal{RT}$. C'est un fichier au format de la librairie PIRATE.

Le dernier argument de la ligne de commande est le nom du fichier d'entrée. Il s'agit dans

le cas présent du fichier « arctg_hex.pie ». Ce fichier doit être un fichier au format de la librairie **PIRATE**. Il doit contenir la géométrie maillée, le maillage initial et le champs solution sur lequel on veut adapter le maillage (champs spécifié par l'option « -H »).

Finalement, il est utile de noter que tel quel, **OO**$\mathcal{RT}$ fera une adaptation de maillage en mode structuré. Il s'agit du comportement par défaut. Pour adapter en non structuré, il faut spécifier l'option sans argument « -N ».

I.2 Fichier de configuration

Le fichier de configuration utilisé dans la ligne de commande présentée à la section précédente est le suivant :

```
Convertisseur Tout() =
{
   borneMinimumEuclidienneDeLaMetrique   0.001;
   borneMaximumEuclidienneDeLaMetrique   0.2;
   etirementMaximumDeLaMetrique          100;
   traitementDeLEtirementMaximum PRESERVE_LAMBDA_MIN;
   longueurCibleDUneAreteDansLaMetrique  0.05;
};
CritereGeometrique Tout() =
{
   nombreDeChiffresSignificatifsDansLesFichiers 7;
};
CritereUsager Tout() =
{
   longueurMinimumEuclidienneDesAretes  0.001;
   longueurMaximumEuclidienneDesAretes  0.2;
   formeMinimumEuclidienneDesElements   0.001;
   distorsionMaximumEuclidienneDUneFace 0.999;
   aretesTransversalesInterdites        oui;
};
CritereMetrique Tout() =
{
   maximumDIterationsGlobalesEnNonStructure 3;
   critereDeConvergenceEnNonStructure       0.1;
   facteurDeRelaxationDansLeDeplacement     1.0
   critereDArretDuDeplacement               0.0090;
   maximumDIterationsDeDeplacement          400;
   fudgeFactorDansLeDeplacement             0.3;
   nombreDIterationsLinearisees             0;
```

```
    maximumDIterationsDeRetournement          5;
    coefficientMultiplicateurDeLEcartTypeDansLaMetrique 1;
    maximumDIterationsDeRaffinement           5;
    seuilDeRaffinement                        1.5;
    maximumDIterationsDeDeraffinement         5;
    seuilDeDeraffinement                      0.5;
    niveauDePrecisionDans[...]VolumesMetriques -1;
    precisionDans[...]DesAretesDansLaMetrique 0.01;
    nombreMaximumDeSubdivisionsDansLe
    RombergLorsDeLIntegrationDesAretesDansLaMetrique 3;
    integrationDesAretesTientCompteDeLaGeometrie  oui;
};
```

Dans ce fichier, il existe une multitude de données qui peuvent être ajustée de façon à contrôler l'adaptation de maillage. Ces données sont regroupées par catégorie.

D'abord, la catégorie « Convertisseur Tout() » permet de définir quelques contraintes à respecter sur la métrique et permet de spécifier la taille cible des arêtes dans cette métrique, ce qui permettra d'ajuster les paramètres de construction de la métrique.

La catégorie « CritereGeometrique Tout() » permet de définir la précision des données contenues dans le fichier d'entrée.

La catégorie « CritereUsager Tout() » permet à l'usager de définir des contraintes sur la longueur des arêtes afin de contrôler certains aspects du maillage.

Finalement, la catégorie « CritereMetrique Tout() » permet de définir les différents éléments en lien avec les processus itératifs. On peut y définir entre autres le nombre maximal d'itération et le critère d'arrêt pour le déplacement de sommets.

ANNEXE II

Manuel d'utilisation de IP-OO$\mathcal{RT}$

Les options en rapport direct avec l'adaptation comme le nom de la variable contenant le champ à partir duquel la métrique est construite, le nom du fichier de sortie et autres sont exactement les mêmes que dans OO$\mathcal{RT}$. En fait, IP-OO$\mathcal{RT}$ repose sur OO$\mathcal{RT}$ par un mécanisme d'héritage et ces fonctions sont traitées par la partie propre à OO$\mathcal{RT}$. IP-OO$\mathcal{RT}$ définit toutefois des options supplémentaires propres à l'adaptation parallèle.

La première option propre à IP-OO$\mathcal{RT}$ est « -f ». Cette option permet de spécifier le numéro de la face topologique du domaine qui sera utilisée pour la construction de la première couche de sommets. Tous les sommets situés sur cette face seront placés sur la couche initiale. Ce numéro doit correspondre à une face topologique contenue dans la géométrie.

La deuxième option propre à IP-OO$\mathcal{RT}$ est « -e ». Cette option permet à l'usager de spécifier le nombre de couches internes à mettre dans chaque bloc dans l'algorithme de partitionnement statique.

La troisième et dernière option propre à IP-OO$\mathcal{RT}$ est « -r ». Cette option permet de spécifier la fréquence de partitionnement. En fait, en donnant l'argument « x : y » à l'option, cela signifie que le domaine doit être repartitionné à toutes les « x » itérations à partir de l'itération « y ».

Les options deux et trois ne peuvent pas être choisies simultanément. La deuxième correspond au cas où le repartitionnement dynamique est désactivé, c'est-à-dire qu'on choisit l'algorithme de partitionnement statique, alors que la troisième permet à l'usager de

spécifier l'utilisation du repartitionnement dynamique.

Ensuite, mentionnons que le lancement de IP-OORT se fait en utilisant la commande « mpirun » tel que définit dans le standard de MPI. Cette commande permet de lancer une application utilisant MPI.

La ligne de commande suivante est un exemple pour le cas test M1 pour lequel l'algorithme de partitionnement statique est utilisé avec un nombre de couche interne égale à 0 :

```
mpirun -np $NP poort.exe -H var -c oort.cfg -f 102 -e 0
                         -o poort.$NP.pie arctg_hex.pie
```

La ligne de commande suivante est un exemple pour le cas test M1 pour lequel l'algorithme de repartitionnement dynamique est utilisé avec un repartitionnement à toutes les 5 itérations, le premier repartitionnement ayant lieu à la deuxième itération :

```
mpirun -np $NP poort.exe -H var -c oort.cfg -f 102 -r 5:2
                         -o poort.$NP.pie arctg_hex.pie
```

ANNEXE III

Utilisation de Open PBS

Tous les tests lancés dans le cadre du projet l'ont été à l'aide du système de gestion de queues de travail Open PBS[1]. Ce système permet de gérer des queues de travail. Afin de soumettre une tâche, il suffit de créer un script et de le soumettre au gestionnaire des tâches. Une partie des scripts utilisés pour les tests de $\mathbb{P}$-**OO**$\mathcal{RT}$ est présentée dans cette section d'annexe.

Ainsi, pour chacun des cas tests, un script PBS a été créé pour **OO**$\mathcal{RT}$ et $\mathbb{P}$-**OO**$\mathcal{RT}$. Par exemple, les deux scripts suivants ont été conçus pour le cas test M1 pour afin d'être lancé sur HEYSE. À noter que les scripts pour CHARYBDE sont légèrement différents puisque l'implantation MPI n'est pas la même (MPICH-GM plutôt que LAM).

Script PBS pour le lancement de **OO**$\mathcal{RT}$ sur HEYSE pour le cas test M1 :

```
#!/bin/bash
#PBS -l nodes=1
#PBS -j oe
#PBS -m n
NP=`wc -l $PBS_NODEFILE | awk '{print $1}'`
cd $PBS_O_WORKDIR

lamboot $PBS_NODEFILE
oort.exe -H var -c oort.cfg -o oort.pie arctg_hex.pie
lamhalt
```

Script PBS pour le lancement de $\mathbb{P}$-**OO**$\mathcal{RT}$ sur HEYSE pour le cas test M1 :

```
#!/bin/bash
```

[1] http ://www.openpbs.org/

```
#PBS -l nodes=1
#PBS -j oe
#PBS -m n
NP=`wc -l $PBS_NODEFILE | awk '{print $1}'`
cd $PBS_O_WORKDIR

lamboot $PBS_NODEFILE
mpirun -np $NP poort.exe -H var -c poort.cfg -f 102 -e 0
                         -o poort.$NP.pie arctg_hex.pie
lamhalt
```